NPS ARCHIVE
1998.12
PHILLIPS, D.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MATHEMATICAL MODELING USING MATLAB

by

Donovan D. Phillips

December 1998

Thesis Advisor:	Maurice D. Weir
Second Reader:	Bard K. Mansager

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1998	Master's Thesis

4. TITLE AND SUBTITLE
MATHEMATICAL MODELING USING MATLAB

5. FUNDING NUMBERS

6. AUTHOR(S)
Phillips, Donovan D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Mathematical modeling forms a bridge between the study of mathematics and the application of mathematics with the intent of explaining or predicting real world behavior. In their book *A First Course in Mathematical Modeling*, Frank R. Giordano, Maurice D. Weir, and William P. Fox provide an introduction to the entire modeling process. Model verification, an important step in the modeling process, often requires the analysis of vast amounts of data, making computational support essential. *Mathematical Modeling Using MATLAB* acts as a companion resource to *A First Course in Mathematical Modeling* with the goal of guiding the reader to a fuller understanding of the modeling process through the employment of MATLAB's powerful computational capabilities. In it, the reader is led through a series of examples, each building upon the previous, which apply MATLAB's computational power to various modeling scenarios. While not intended as a text in modeling, *Mathematical Modeling Using MATLAB* is a useful resource for the novice modeler interested in tackling problems too large to be performed manually.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Mathematical Modeling, Discrete Dynamical Systems, Proportionality, Model fitting	132
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

MATHEMATICAL MODELING USING MATLAB

Donovan D. Phillips
Captain, United States Army
B.S., United States Military Academy, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
December 1998

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

ABSTRACT

Mathematical modeling forms a bridge between the study of mathematics and the application of mathematics with the intent of explaining or predicting real world behavior. In their book *A First Course in Mathematical Modeling*, Frank R. Giordano, Maurice D. Weir, and William P. Fox provide an introduction to the entire modeling process. Model verification, an important step in the modeling process, often requires the analysis of vast amounts of data, making computational support essential. *Mathematical Modeling Using MATLAB* acts as a companion resource to *A First Course in Mathematical Modeling* with the goal of guiding the reader to a fuller understanding of the modeling process through the employment of MATLAB's powerful computational capabilities. In it, the reader is led through a series of examples, each building upon the previous, which apply MATLAB's computational power to various modeling scenarios. While not intended as a text in modeling, *Mathematical Modeling Using MATLAB* is a useful resource for the novice modeler interested in tackling problems too large to be performed manually.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1
B. INTRODUCTION TO MATLAB 2
1. Structure 2
2. Functions and Operations 9
3. Workspace 10
4. Variables 11
5. Online Help 12

II. MODELING WITH DISCRETE DYNAMICAL SYSTEMS 15

A. INTRODUCTION 15
B. EXAMPLE 1: THE ARMS RACE 15
1. Scenario 15
2. Modeling the Dynamics of the Arms Race 16
3. Plotting 21
C. EXAMPLE 2: POPULATION GROWTH IN A YEAST CULTURE 27
1. Scenario 27
2. An Initial Model 28
3. Model Refinement 31
4. Numerical Solution for the Refined Model 33
5. Bifurcation and Chaos 34
D. EXAMPLE 3: LANCHESTER'S SQUARE LAW DISCRETE MODEL 41
1. The Model 42
2. A Numerical Example 42

III. MODELING USING PROPORTIONALITY 47

A. EXAMPLE 1: VEHICULAR STOPPING DISTANCE 47
1. Initial Model 47
2. Model Refinement 49
B. EXAMPLE 2: A BASS FISHING DERBY 56
1. Initial Model 56
2. Model Refinement 60

IV. MODELING FROM DISCRETE DATA 63

A. MODEL FITTING 64
1. Vehicular Stopping Distance -- Another Approach 65
2. Residual Plots 66
3. Using MATLAB's Polyfit Function 68
B. EMPIRICAL MODELING 70
1. High-Order Polynomial Models 71
2. Low-Order Polynomial Models 75
3. Cubic Spline Models 84

V. SIMULATION MODELING 87

A. RANDOM NUMBER GENERATION IN MATLAB 87

B. SIMULATING DETERMINISTIC BEHAVIOR 88
1. Area Under a Curve *88*
2. Volume Under a Surface *91*
C. SIMULATING PROBABILISTIC BEHAVIOR 92
1. Tossing a Fair Coin *92*
2. The Roll of a Fair Die *94*

VI. LINEAR PROGRAMMING 97

A. GEOMETRIC SOLUTIONS 98
B. TABLEAU SIMPLEX METHOD 102

VII. CONCLUSION 111

APPENDIX 113

INDEX 119

LIST OF REFERENCES 121

INITIAL DISTRIBUTION LIST 123

I. INTRODUCTION

A. BACKGROUND

Mathematical modeling is the science of explaining and predicting (to the extent possible) "real world" behavior through the application of mathematics. Before we would use a potential model, however, we would first want to verify, or validate, the model to ensure it makes sense and that it answers the question we wish answered. This validation often requires detailed analysis of large amounts of data using a combination of modern computer hardware and appropriate software. MATLAB (matrix laboratory) is one such software package.

MATLAB integrates computation, visualization, and programming in an easy-to-use environment expressing problems and solutions in familiar mathematical notation. This makes it a useful tool in the process of mathematical modeling. MATLAB is capable of running on several platforms, including UNIX systems and personal computers (PCs). In this thesis I describe the use of MATLAB, Version 5.2 (the latest version as of this printing) operating in the PC environment; however, the methods are virtually identical from platform to platform. A complete description of this release of MATLAB can be found in Mastering MATLAB 5: A Comprehensive Tutorial and Reference (1).

B. INTRODUCTION TO MATLAB

1. Structure

MATLAB is an interactive system whose basic data element is a rectangular matrix (note: row and column vectors and even scalars are simple forms of rectangular matrices). A variety of mathematical operations can be easily performed on these data arrays to achieve whatever results are desired. MATLAB offers two environments in which these operations can be performed: the **Command Window** and **script files.** We will discuss the Command Window first.

a) The Command Window

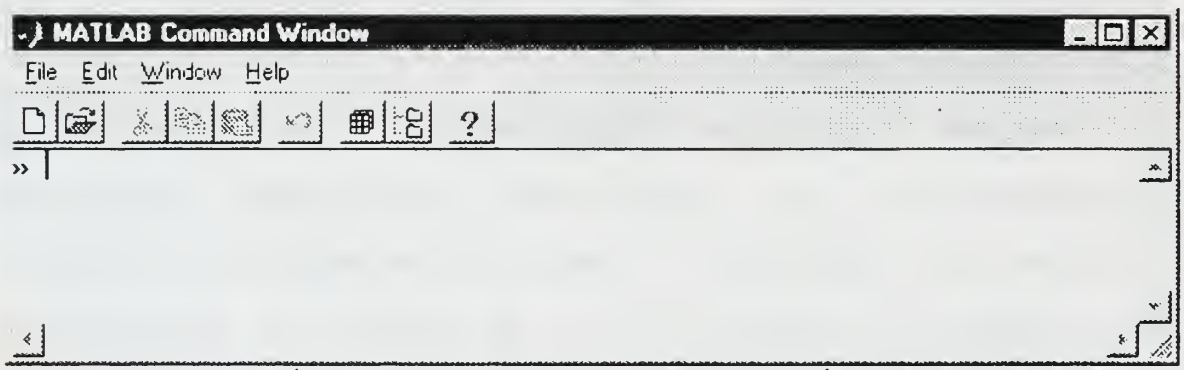

Figure I.1. MATLAB Command Window

The Command Window (shown in Figure I.1) is displayed when MATLAB is first invoked. Commands are entered at the ">>" symbol (the command prompt) and are executed when the [Enter] key is depressed. For example, to sum the numbers 10, 20, 30, and 40, one would enter the following at the command prompt:

```
10 + 20 + 30 + 40  [Enter]
```

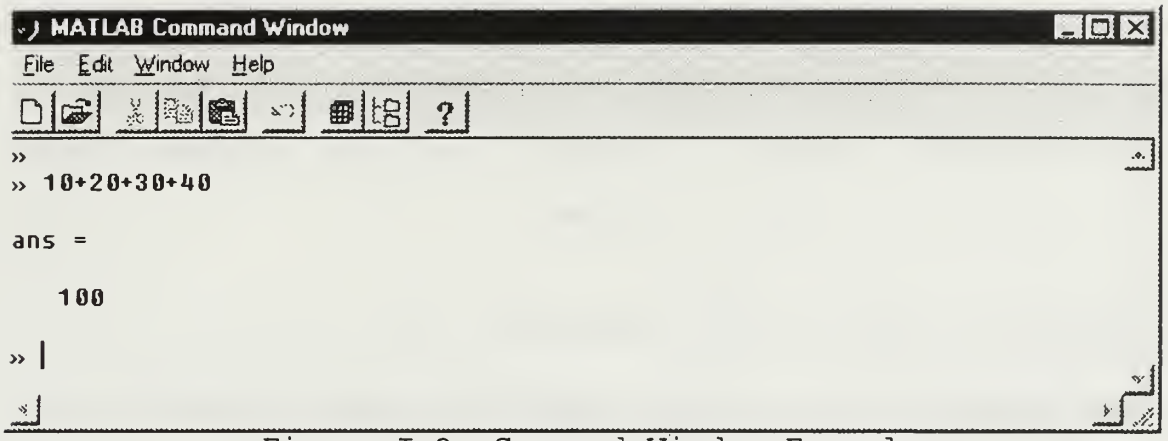

Figure I.2. Command Window Example

Note the term "ans" following the entry at the command prompt in Figure I.2. MATLAB assigns the result of a computation to this temporary variable, which is overwritten by succeeding unassigned computations. Alternatively, one could assign this expression to a *variable* and then solve for it:

```
» x=10+20+30+40
x =
   100
```

The variable *x* takes on the value of the expression, and subsequent operations can then be performed on *x*. For example:

```
» x=10+20+30+40
x =
   100
» y=sqrt(x)
y =
    10
```

Here we used a MATLAB **function**, i.e. *sqrt*, to obtain the desired result. We could just as easily have used an **operation** (raising a number to a power) to accomplish the same thing:

```
» x=10+20+30+40
x =
   100
» y=x^(1/2)
y =
    10
```

A more thorough discussion of functions and operations will be provided shortly. First, we turn our attention to the MATLAB script file environment.

b) Script Files

For simple problems, the Command Window provides a fast and efficient way to enter data and produce solutions. However, as problems become more complex and the need arises to change the value of one or more variables and reevaluate a series of commands, typing these commands at the MATLAB prompt quickly becomes tedious and unappealing. MATLAB solves this problem by enabling you to write (and store) text files containing the desired sequence of commands that can be executed simply by entering the file name in the Command Window. MATLAB executes the commands exactly as if you had typed them at the command prompt. These files are commonly referred to as **M-files** since the file name must end with the extension **.m** ("*dot m*"), as in **example.m**. Because

the file is stored, you can run it again and again (making changes if desired) by merely typing its name at the command prompt. MATLAB provides a text editor, known as the MATLAB Editor/Debugger, for the purpose of writing and editing M-files.

(1) Using M-files. To open the MATLAB Editor/Debugger from the Command Window, click on **File**, then **New**, followed by **M-file**.

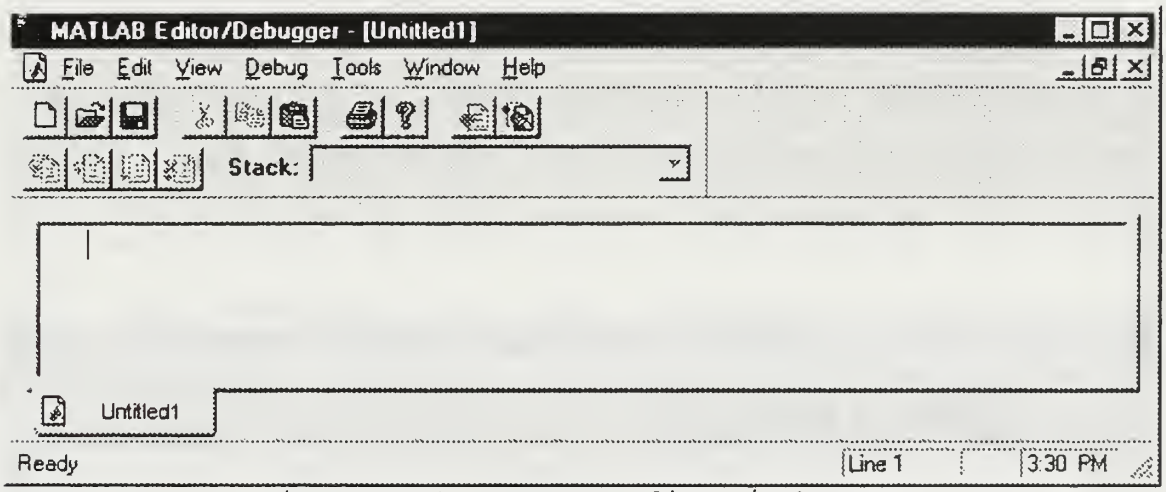

Figure I.3. MATLAB Editor/Debugger

In addition to acting as a text editor, the MATLAB editor/debugger allows you to execute M-files directly from its window (without having to switch to the Command Window), and assists in debugging code. What follows in Figure I.4 is a simple example of an M-file written in the MATLAB editor. The example multiplies each number in a sequence by 3.

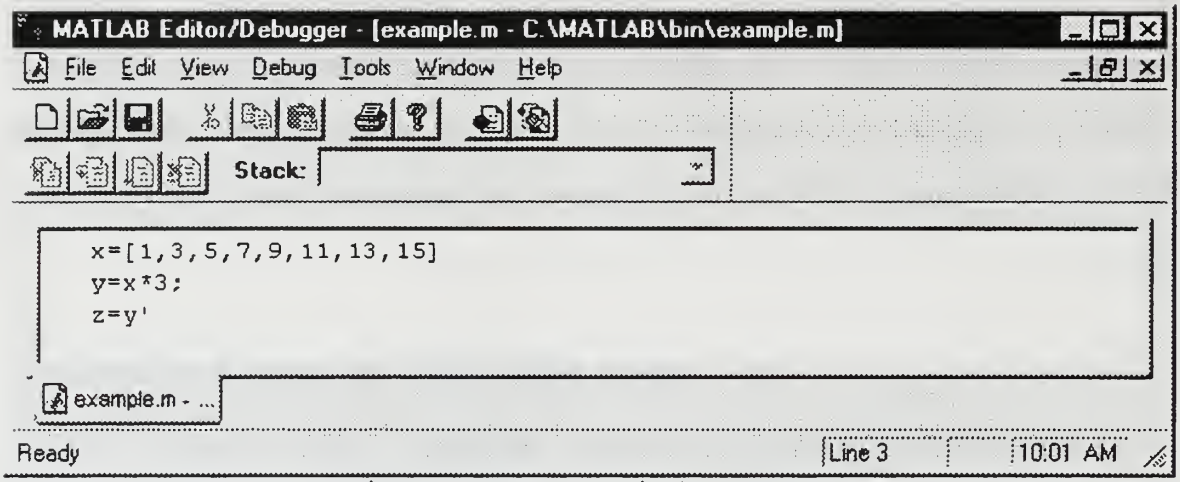

Figure I.4. M-file Example

You can execute this M-file in one of two ways: switch to the Command Window and enter the name of the file (in this case, **example.m**) at the command prompt, or select **Tools** from the MATLAB editor "pull-down" menu and click on **Run**. Either method will net the same results, shown in Figure I.5.

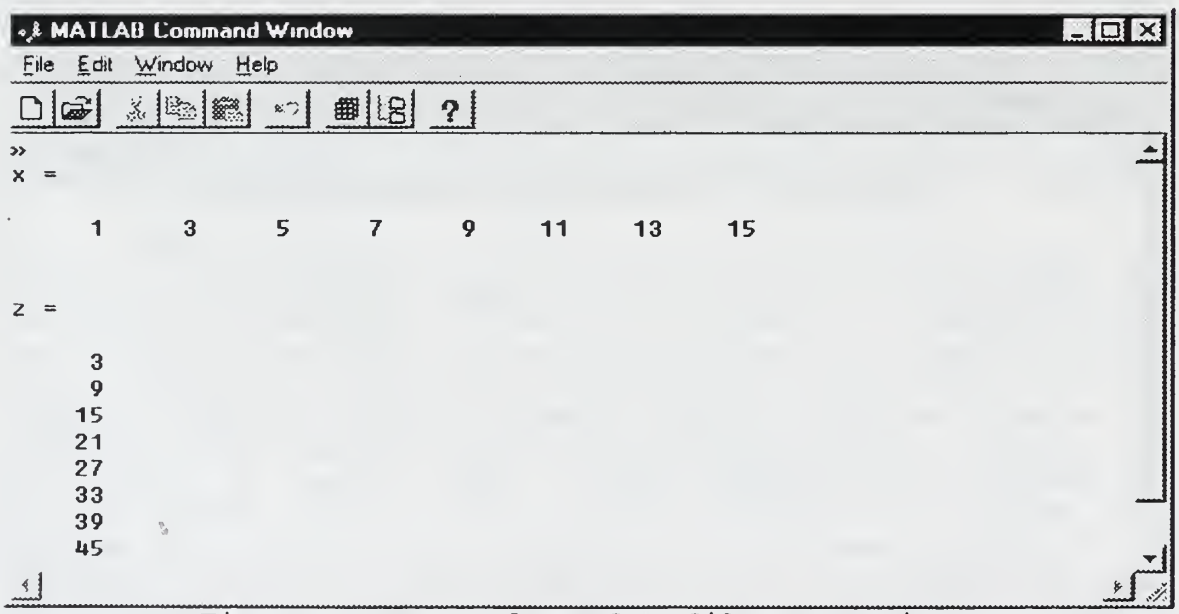

Figure I.5. Results of M-file Execution

The commands in the M-file are executed exactly as if they had been typed directly in the Command Window. As we will see shortly, M-files are very useful in modeling applications.

We have introduced a few new ideas with the above example that deserve mention. First, notice in Figure I.4 that the variable *x* represents a **vector**. The square brackets **[ ]** are used to denote a sequence of numbers as a vector, with commas separating each element. Next, notice the use of the **semicolon** following the line *y = x * 3* in Figure I.4. The semicolon acts to *suppress* the display in the Command Window of the results of the command preceding it. Also, notice the absence, due to the use of the semicolon, of any reference to the variable *y* in Figure I.5. The semicolon is a very powerful tool that greatly simplifies the analysis of large arrays of data by suppressing from view the results of intermediate calculations that are not necessary to see. Finally, note the use of the **apostrophe** following the line *z = y*. The apostrophe serves to **transpose** the vector **y**, changing it from a row vector to a column vector. This distinction is important when manipulating data arrays.

(2) File Management Involving M-files.

One may receive an error message, like the following, when attempting to run an M-file either from the Command Window or from the MATLAB Editor (the M-file in this case is called **armsl.m**):

```
» armsl
??? Undefined function or variable 'armsl'.
```

If this situation occurs, then one of two things has happened (assuming you typed the M-file name correctly): either the file was not previously saved, or (more likely) the directory in which the file is stored is not included in the MATLAB search path. The first problem is easily remedied; the second requires adding your M-file storage directory to the MATLAB search path. To do this, click on **File** in the Command Window, and then on **Set Path**. This causes the MATLAB Path Browser window to become active. Now add your M-file storage directory to the search path by either typing the directory name into the box marked "Current Directory" or by using the Browse button to find and highlight the appropriate directory. Once either of these actions is done, add this directory to the path by clicking on **Path**, followed by **Add to Path**. Once this is done, save the path by clicking on **File**, then **Save Path**. Once the path is saved, you can close the Path Browser window and return to the Command Window. M-files stored in

this directory may now be run from both the Command Window and the MATLAB Editor.

2. Functions and Operations

MATLAB offers the following basic arithmetic operations:

Operation	Symbol	Example
Addition	+	2+8
Subtraction	-	16.3-14
Multiplication	*	8*19
Division	/ or \	15/3=3\15
Power	^	5^3

Table I.1. MATLAB Operations

The following table shows a sample of the functions offered by MATLAB:

	Common Functions
abs(x)	Absolute value
cos(x)	Cosine
exp(x)	Exponential: e^x
log(x)	Natural logarithm
log10(x)	Base 10 logarithm
sin(x)	Sine
sqrt(x)	Square root
tan(x)	Tangent

Table I.2. Common Functions

Other functions will be introduced as needed throughout this thesis.

3. Workspace

MATLAB remembers all commands entered in the Command Window as well as the value of any variables you create. These commands and variables reside in the **MATLAB Workspace** and can be recalled on demand. For example, to find the current value of the variable *x*, simply enter its name at the command prompt and its value will be displayed:

```
» x
x =
     1    3    5    7    9   11   13   15
```

If you forget the name of some or variables in use, the command **whos** results in a listing of all the variables created in the current session:

```
» whos
  Name        Size              Bytes  Class
  x           1x8                  64  double array
  y           1x8                  64  double array
  z           8x1                  64  double array
```

The **size** of the variable indicates whether it is a scalar, vector, or matrix.

MATLAB allows you to recall previously entered commands by using the **up arrow** on your keyboard. Pressing this key once recalls the most recent command; each successive pressing results in the next most recent command, allowing you to scroll backward through the commands you have entered. The **down arrow** lets you scroll forward in similar fashion. Once you have scrolled to the desired command, you

may re-execute the command or edit it prior to execution. Hitting the enter key executes the command.

4. Variables

a) Naming Variables

MATLAB has very specific rules about naming variables. A variable name:

- must be a single word containing no spaces;
- is case sensitive; i.e., ITEMS, Items, items, and itEms are all **different** MATLAB variables;
- can contain up to 19 characters;
- must start with a letter, followed by any number of letters, digits, or underscores.

b) Redefining Variables

Variables may be redefined as desired. For example,

```
» apples=4;
» oranges=7;
» applesandoranges=apples+oranges
applesandoranges =
    11
» apples=7
apples =
     7
» applesandoranges
applesandoranges =
    11
```

Notice that changing the value of *apples did not* cause the value of *applesandoranges* to change. Unlike a spreadsheet, MATLAB does not recalculate the number *applesandoranges* based on the new value of *apples*. MATLAB performs calculations based on the information available at the time that the command is executed. In order to update the total fruit count, one must reissue the command:

```
» applesandoranges=apples+oranges
applesandoranges =
    14
```

c) Deleting Variables

MATLAB variables can be irrecoverably deleted from the workspace using the **clear** command. For example,

```
» clear oranges
```

deletes the variable *oranges*; the command

```
» clear
```

deletes all of the variables in the workspace. Needless to say, the **clear** command should only be used with extreme caution.

5. Online Help

Should you need it, MATLAB offers an extensive library of help capabilities that are available in three forms: the MATLAB commands **help** and **lookfor** and interactively using the **Help Desk** from the pull-down menu.

a) The help Command

If you know the name of a MATLAB function, the help command is the simplest way to get information about that function. For example, for help on the Absolute Value function, we would type:

Use lower case for all MATLAB functions.

```
» help abs
 ABS    Absolute value.
      ABS(X) is the absolute value of the elements of X.
      When X is complex, ABS(X) is the complex modulus
      (magnitude) of the elements of X.
See also SIGN, ANGLE, UNWRAP.
```

It is important to note at this point that, just as with variables, MATLAB distinguishes between upper and lower case characters in function names. Notice that the command **ABS** is capitalized in the first line after the command prompt. **This is for readability purposes only**. You must always call MATLAB functions using **lower case** characters. Attempting to call functions using upper case characters will result in error messages like the following:

```
» SQRT(2)
??? Undefined variable or capitalized internal function
SQRT; Caps Lock may be on.
```

The **help** command works well provided you know the name of the function you desire information on. When this is not the case, the next command may assist you.

b) The lookfor Command

The **lookfor** command provides help by searching all MATLAB files for the key word you provide. For example,

```
» lookfor cosine
ACOS    Inverse cosine.
ACOSH   Inverse hyperbolic cosine.
COS     Cosine.
COSH    Hyperbolic cosine.
TFFUNC time and frequency domain versions of a cosine
modulated Gaussian pulse.
```

provides a list of all the functions that contain the key word cosine.

c) The Help Desk

Perhaps the most extensive and user-friendly way to access help in MATLAB is the new (with version 5.2), interactive, Hypertext Markup Language- (HTML-) based **Help Desk**. The Help Desk is accessed from **Help** on the pull-down menu and requires an Internet Web browser, such as Internet Explorer or Netscape, to access. The use of hypertext allows for easy access to any topic by simply pointing and clicking with a mouse. The novice MATLAB user would benefit from a look at the "Getting Started" section, which describes all the basic functions and how they are used.

II. MODELING WITH DISCRETE DYNAMICAL SYSTEMS

A. INTRODUCTION

In Chapter I, we discussed some basic essentials for using MATLAB. Here, we delve a bit deeper into more sophisticated features offered by MATLAB. New topics introduced in this chapter include **data manipulation**, **looping**, and **plotting**, which are used to model change via **discrete dynamical systems**. We present these concepts using several examples of dynamical systems taken from A First Course in Mathematical Modeling by Giordano, Weir, and Fox (2).

B. EXAMPLE 1: THE ARMS RACE

1. Scenario

Section 1.1 of (2) outlines in detail the scenario for this model. Following is a brief summary:

Two countries, **X** and **Y**, are involved in an arms race. We wish to model the advancement of arms proliferation in terms of the number of missiles each country has at a given time. To do this, we need to make the assumption that each country follows a strategy of deterrence that requires it to have a given number of weapons to deter the enemy even if the enemy has no weapons. The strategy of each country is to increase its arms inventory by some percentage of its enemy's arsenal each time the enemy adds weapons to its

inventory. For example, suppose Country Y feels it needs 120 weapons to deter the enemy. Further, for every two weapons possessed by Country X, Country Y feels it needs to add one additional weapon to ensure 120 weapons will remain after a potential strike by Country X. It follows that the number of weapons needed by Country Y (y weapons) as a function of the number of weapons it thinks Country X has (x weapons) is

$$y = 120 + \frac{1}{2}x \tag{2.1}$$

Now suppose Country X is following a similar strategy. It determines it needs 60 weapons even if Country Y has none. Further, for every three weapons it thinks Country Y possesses, Country X feels it must add one weapon. Thus the number of weapons needed by Country X as a function of the number of weapons it thinks Country Y has is

$$x = 60 + \frac{1}{3}y \tag{2.2}$$

We will now look at the dynamic progression of the arms race.

2. Modeling the Dynamics of the Arms Race

Suppose initially that Countries Y and X do not think the other side has arms. Following their strategy, they build 120 weapons and 60 weapons respectively. Now assume

each has perfect intelligence (i.e., each knows the exact number of weapons the other has built). The result is a dynamical progression of the arms race. At each stage, each country adjusts its inventory based on the size of the enemy's arsenal during the *previous* stage. To model this situation in MATLAB, we use the MATLAB Editor to write an M-file:

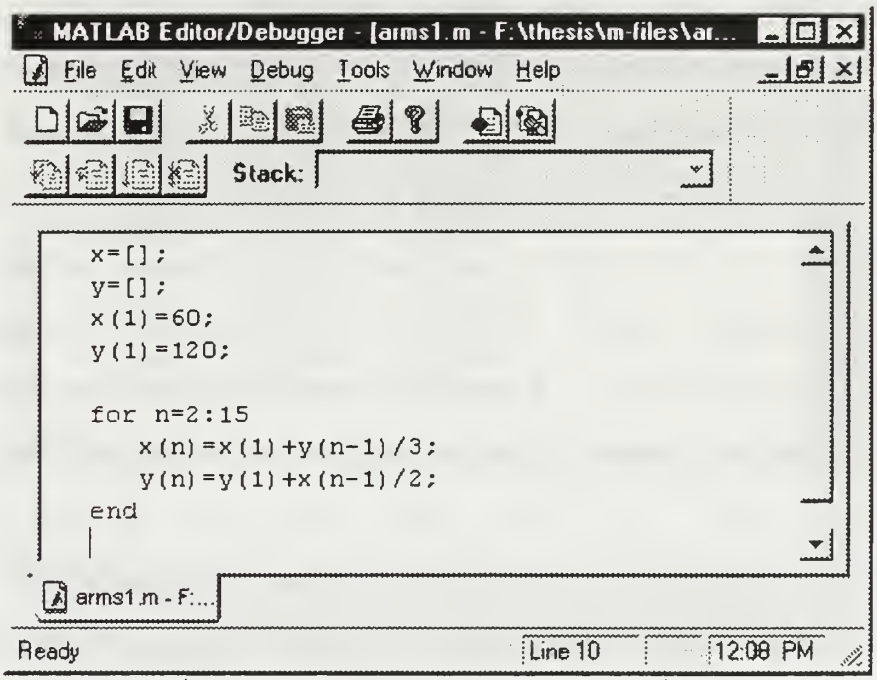

Figure II.1. Arms Race M-file

Before we look at the results, let's discuss the new MATLAB concepts introduced in Figure II.1. First, notice the expressions *x=[]* and *y=[]*. These commands serve to assign temporarily empty vectors (*row* vectors by default) to the variables *x* and *y*, which represent the size of each country's arsenal at each stage. This is done with the

intent of assigning elements to these vectors at a later time, which can be done in various ways. Note how this differs from *explicitly* defining the elements of a vector as demonstrated in Figure I.4. Next, we see that we can assign elements to a vector individually with expressions like *y(1) = 120*. This assigns the value 120 as the *first* element of *y*.

Now that we have our vectors (or *data arrays*) **initialized**, we can now proceed to fill them by successively iterating or **looping** over expressions (2.1) and (2.2). Figure II.1 provides an example of a **for loop**. It consists of an **index** (in this case, *n*, representing the *stage* of the dynamical system) which determines the number of times the loop will be executed, a series of commands executed each time through the loop, and an **end** statement to signal the end of the loop. In this case, the loop starts with an index value of *n = 2* (since the first elements of *x* and *y* have already been assigned). Each time the loop is executed, values for the number of each country's weapons are calculated and assigned as elements of their respective vectors.

The results of running this M-file will be the creation of two row vectors, *x* and *y*, containing elements representing the number of weapons each country has on hand at each of stages one through fifteen. These vectors are stored in the MATLAB Workspace. We can switch to the

Command Window and view this result by entering the following command:

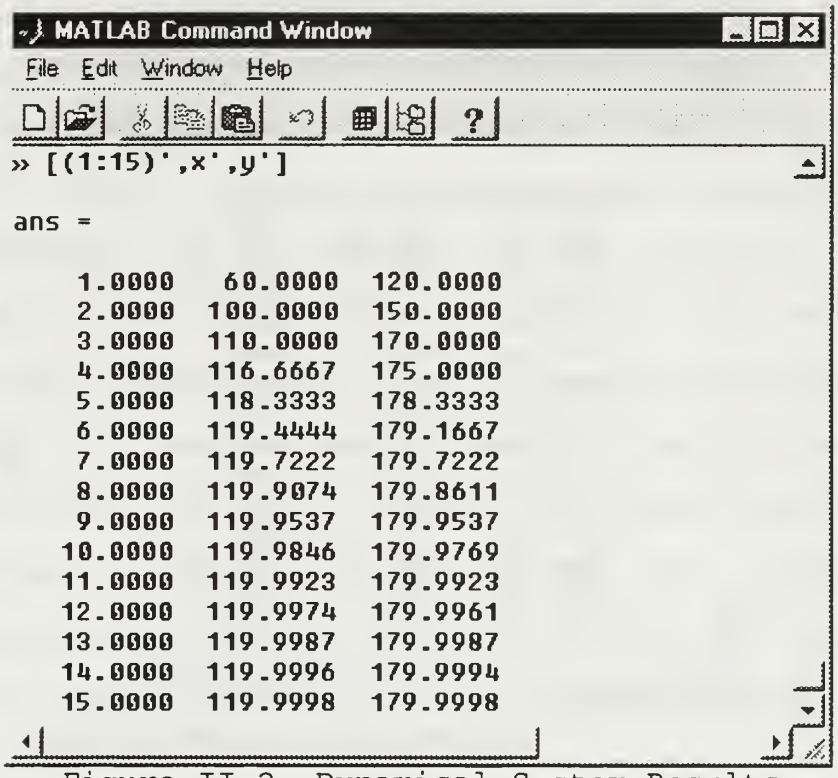

Figure II.2. Dynamical System Results

The expression *[(1:15)',x',y']* can be translated, "Construct a matrix having three columns. In the first column, display the numbers one through fifteen; in the second and third columns, display the transposed vectors *x* and *y*, respectively." The same result could be obtained just as easily with the following alternate commands:

```
» n=1:15;
» [n',x',y']
```

You will find, as we go along, that this situation is not unusual. There are usually more ways than one to obtain a particular desired result in MATLAB.

Let's look at the output itself for a moment. Notice in Figure II.2 that the growth rate in the arms race appears to be decreasing over successive stages. In fact, the size of each country's arsenal appears to be approaching an equilibrium value (120 for country X and 180 for country Y).

Suppose you wanted to know how changing the initial values would effect the equilibrium values. To find out, you need only change the initial values in your M-file and run it again. For example, suppose country X started with 150 weapons and country Y started with 80. To compute the new equilibrium values, you could change your M-file as follows:

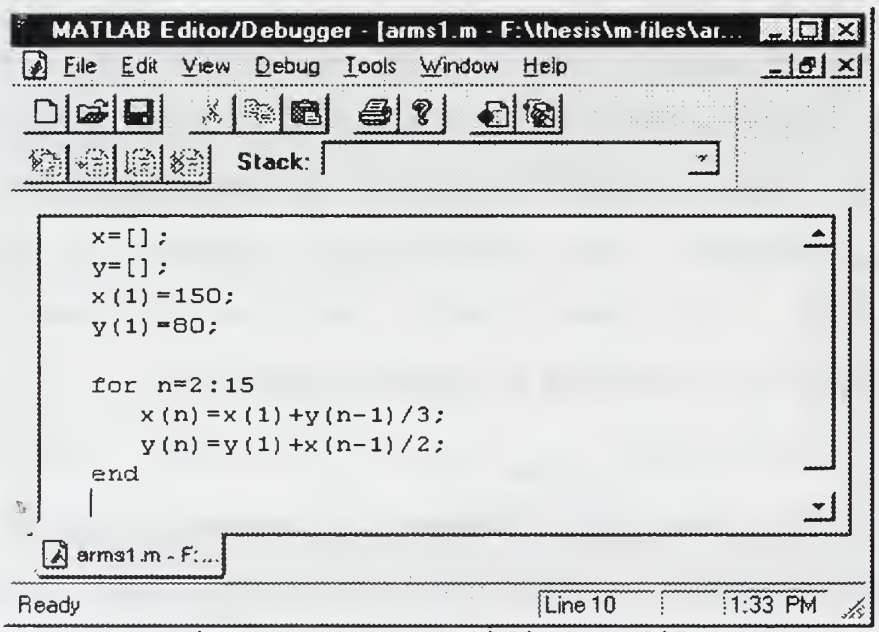

Figure II.3. Modified M-file

The results of running this modified M-file appear below:

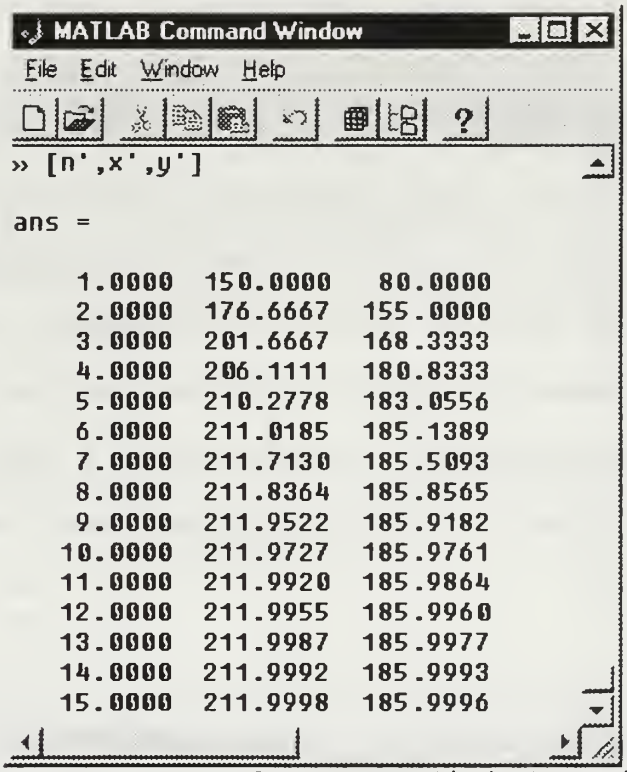

Figure II.4. Results of Modified M-file

It is easy to see that the equilibrium values have changed to 212 for country X and 186 for country Y. This small example of **sensitivity analysis** shows the beauty of using M-files: a few keystrokes (changing the M-file itself) result in the solution of a modified problem that can lead to a more complete understanding of the behavior being modeled.

3. Plotting

Another useful means of analyzing a discrete dynamical system is to plot the data. Simple problems (such as

demonstrated above) can be easily analyzed by displaying the data: trends and key information, such as equilibrium values, are usually obvious. We now address plotting in MATLAB using the Arms Race problem to demonstrate.

MATLAB's basic plotting command, **plot(x,y)**, generates a plot of y versus x, where x and y are vectors of equal length. Before we can plot (for example, the growth in country X's arsenal by stage) we first need a vector to plot against, equal in length to the vector **x**, and representing the various stages. If we let **n** be this stage vector, we adjoin the line

```
n = 1:15;
```

after the "end" statement of our M-file (see Figure II.1) which creates the desired vector. This results in the assignment of a vector containing the elements one through fifteen (in sequence) to the variable **n**. Note that the vectors **x** and **n** are of equal length. Note also that, even though we used the symbol **n** as our **looping index**, we can use the same symbol to represent the stage vector since MATLAB overwrites variables when reassigned (see Chapter I, section B.4.b) and we no longer need the looping index. Now, to produce the desired plot, use the command **plot(n,x)**. This command can either be entered in the command window (after running the modified M-file that creates the vectors **x** and **n**) or adjoined directly to end of the M-file. We use here

the M-file option which, when modified, looks like the following:

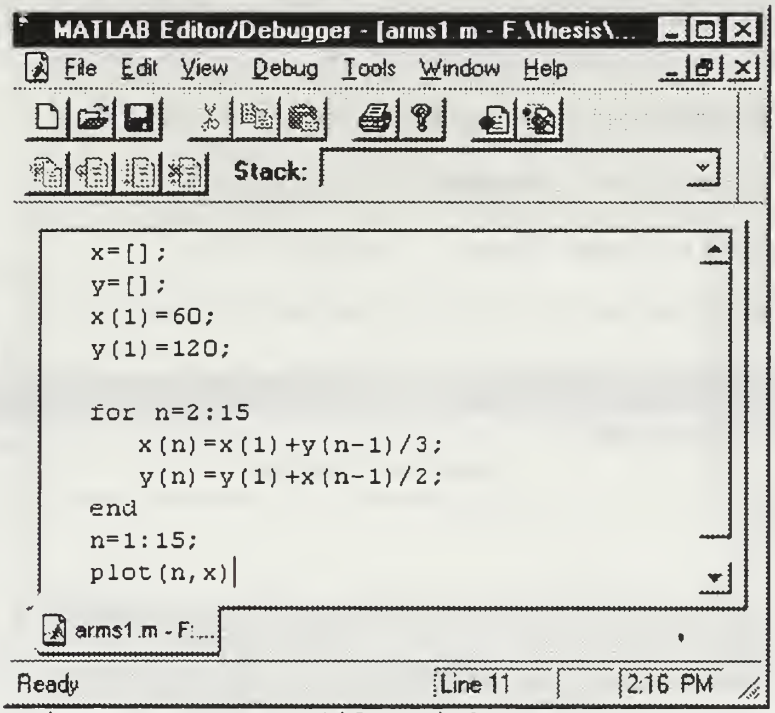

Figure II.5. M-file with Plot Command

Running this M-file produces the graph in Figure II.6.

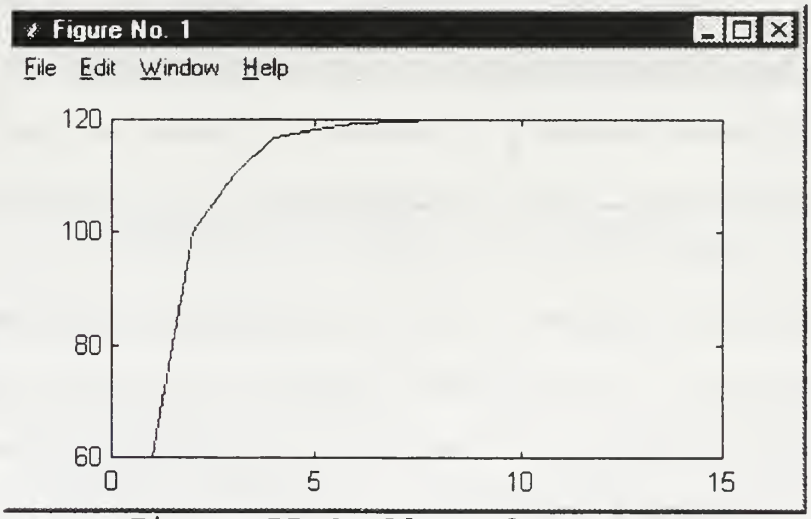

Figure II.6. Plot of x vs. n

Adding axis labels and a title require the use of the commands **xlabel**, **ylabel**, and **title**. To add these features to our plot, simply adjoin the following lines at the end of the updated M-file in Figure II.5:

```
xlabel('n - stages');
ylabel('x - Number of Weapons');
title('Arms Race - Country X');
```

Running this updated M-file produces Figure II.7.

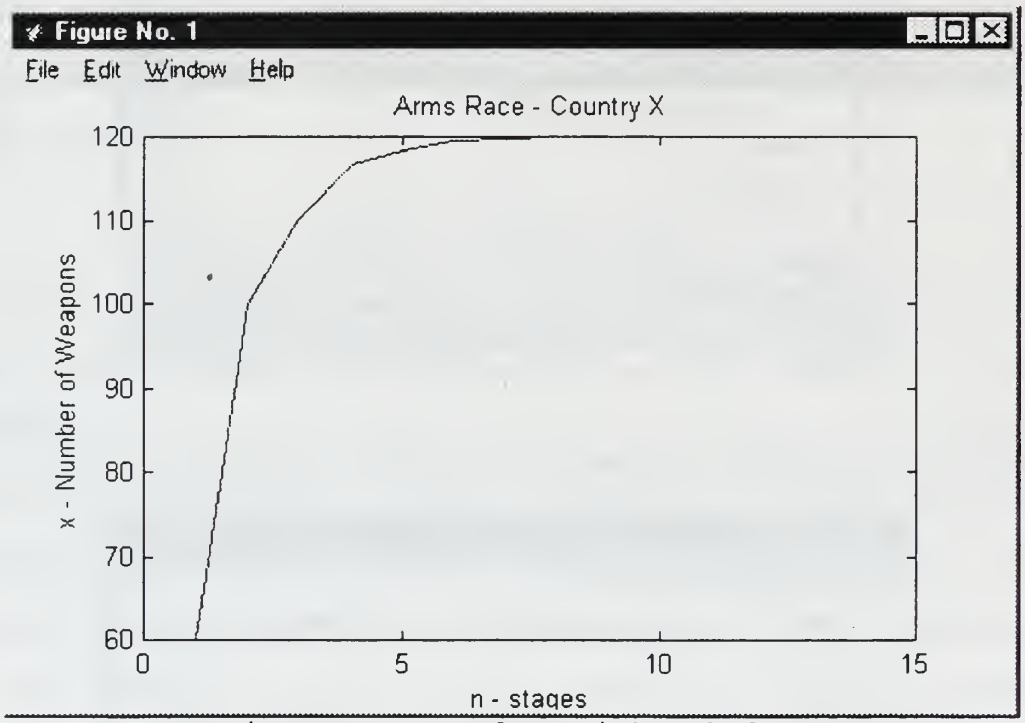

Figure II.7. Plot with Labels

Putting the graphs of both countries on the figure is almost as easy. It requires changing the plot command to the following (again, this can be done either in the command window or in the M-file):

```
plot(n,x,n,y)
```

This modification, along with appropriate label changes, results in Figure II.8.

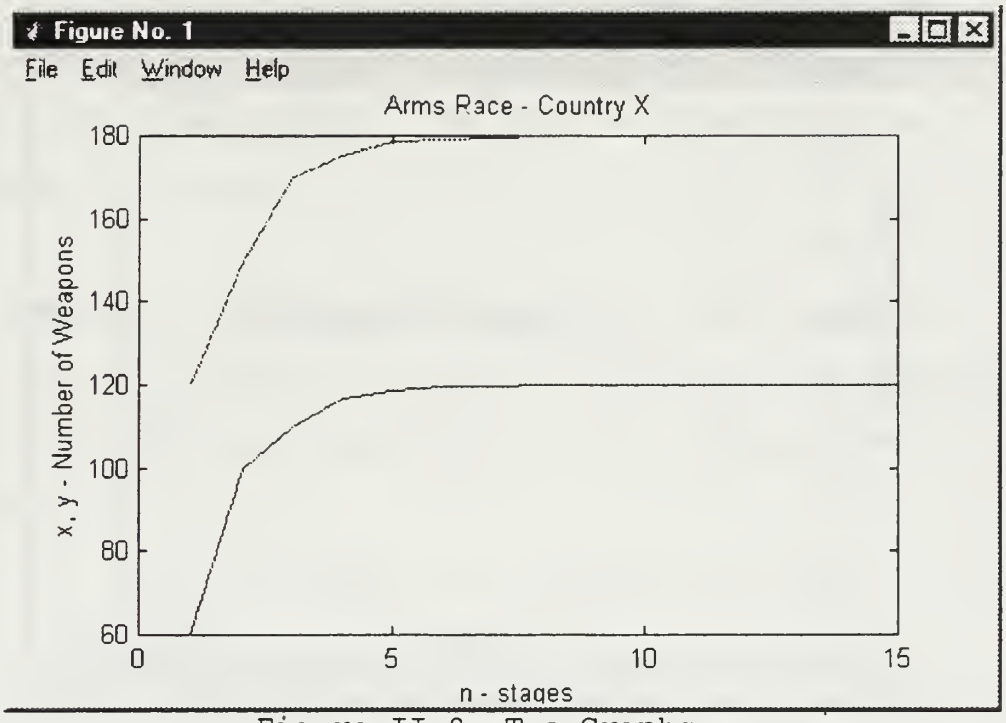

Figure II.8. Two Graphs

However, in black and white, it is difficult to discern which curve represents Country X and which represents Country Y. We need a way to distinguish among several curves in a single figure: the curves must be distinguishable and labeled. This can be accomplished by making one of the curves a dashed line and adding a legend. The commands are as follows:

```
plot(n,x,n,y,'--');
legend('Country X', 'Country Y')
```

The resulting figure follows:

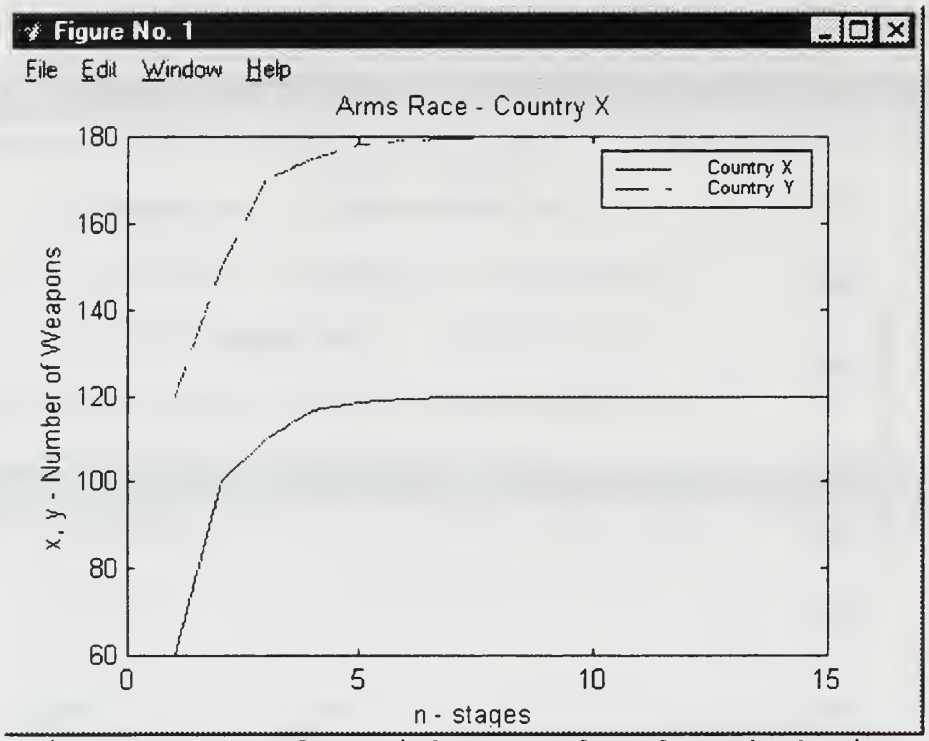

Figure II.9. Plot with Legend and Dashed Line

Notice how the entry '--' in the plot command causes the curve for Country Y to be dashed. This modification is one of many that can be added to a plot command to obtain line effects and colors. Table II.1 lists all such modifiers:

Symbol	Color	Symbol	Line Style
y	yellow	.	point
m	magenta	o	circle
c	cyan	x	x-mark
r	red	+	plus
g	green	*	star
b	blue	-	solid line
w	white	:	dotted line
k	black	-.	dash-dot line
		--	dashed line

Table II.1. Basic Plot Line Types and Colors

It is easy to see the value of plotting data for a discrete dynamical system. MATLAB provides a wide assortment of plot features to enhance the user's ability to analyze dynamical systems. We discuss some of these in the examples that follow.

C. EXAMPLE 2: POPULATION GROWTH IN A YEAST CULTURE

With this example, we look at modeling by approximating change with difference equations. This problem can be found in Section 3.2 of (2).

1. Scenario

Consider the following experimental data regarding the growth of a yeast culture:

Time in hours **n**	Observed yeast biomass p_n	Change in biomass $p_{n+1}-p_n$
0	9.6	8.7
1	18.3	10.7
2	29.0	18.2
3	47.2	23.9
4	71.1	48.0
5	119.1	55.5
6	174.6	82.7
7	257.3	

Table II.1. Yeast Population Data

To make a scatter plot of the growth of the biomass, use the following M-file:

```
x=[9.6 18.3 29 47.2 71.1 119.1 174.6];
y=[8.7 10.7 18.2 23.9 48 55.5 82.7];
plot(x,y,'ko')
xlabel('Biomass')
ylabel('Change in Biomass')
title('Change in Biomass vs. Biomass')
```

Recall that the **'ko'** modifier in the **plot** command causes each data point to be plotted with a black circle (see Table II.1). Running this M-file results in Figure II.10.

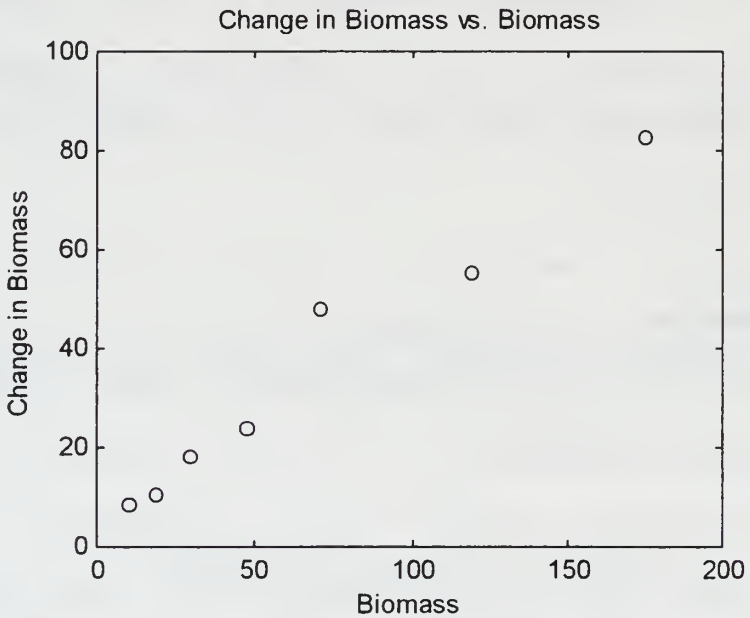

Figure II.10. Yeast Culture Scatter Plot

2. An Initial Model

Although the graph of the data does not lie precisely along a straight line passing through the origin, you can see that such a line can approximate it. We can modify our M-file to produce this line, as follows:

```
x=[9.6 18.3 29 47.2 71.1 119.1 174.6];
y=[8.7 10.7 18.2 23.9 48 55.5 82.7];
```

```
k=rto(x,y);
z=0:max(x);
plot(x,y,'ko',z,k*z,'k')
text(80,30,['slope = ',num2str(k)])
xlabel('Biomass')
ylabel('Change in Biomass')
title('Change in Biomass vs. Biomass')
```

The function **rto(x,y)** performs "regression through the origin" of y on x and outputs the slope of the best-fit line (in the least squares sense) through the origin. This function is **not available in MATLAB**, but can be found in the appendix. In addition, the command **z = 0:max(x)** results in the assignment to the variable **z** of a vector of values starting at zero and ending at the largest value in the vector **x**, with a step size of one. For example, the command

```
a = 1:5
```

results in the output

```
a =
     1     2     3     4     5.
```

Finally, we use the **text** command for the first time here. This command allows you to label items on a graph with text. The numeric arguments that appear in this command represent the coordinates of the location on the graph for the text to begin. The command **num2str** used within the text command converts a number to a text string, which is necessary for inclusion of numerical data in a text expression. The graph produced by this M-file appears in Figure II.11.

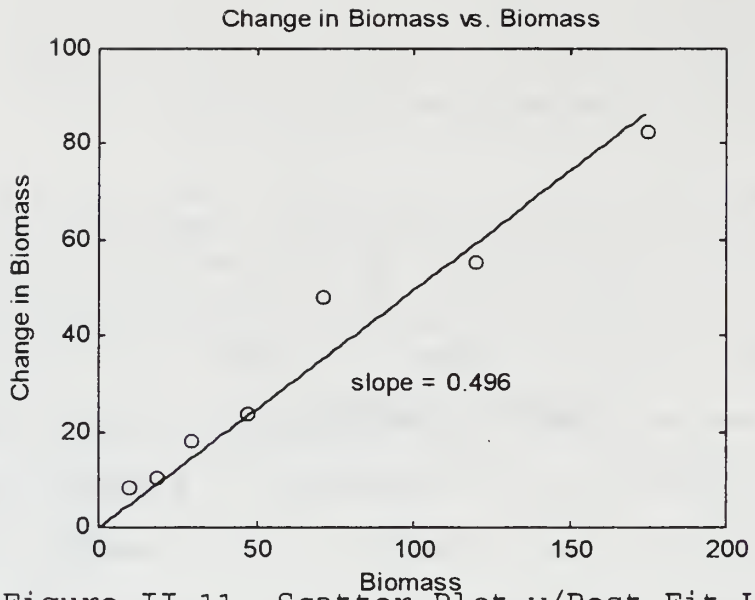

Figure II.11. Scatter Plot w/Best Fit Line

From this information we can derive a proportionality model for population growth:

$$\Delta p_n = p_{n+1} - p_n = 0.496 p_n \tag{2.3}$$

yielding the model

$$p_{n+1} = 1.496 p_n \tag{2.4}$$

to predict future population at the next stage based on population at the current stage. Notice that this model predicts a population that increases without bound, which, in all likelihood, is quite unrealistic. Thus, our model needs some refinement.

3. Model Refinement

Realistically, the availability of critical resources (air, light, food, etc.) in the environment supporting the population tends to limit the population to some maximum level, known as the carrying capacity. Table II.3 shows what actually happens to the yeast culture growing in a restricted environment for time periods beyond those shown in Table II.2.

Time	Observed	Change
0	9.6	8.7
1	18.3	10.7
2	29.0	18.2
3	47.2	23.9
4	71.1	48.0
5	119.1	55.5
6	174.6	82.7
7	257.3	93.4
8	350.7	90.3
9	441.0	72.3
10	513.3	46.4
11	559.7	35.1
12	594.8	34.6
13	629.4	11.4
14	640.8	10.3
15	651.1	4.8
16	655.9	3.7
17	659.6	2.2
18	661.8	

Table II.3. Extended Yeast Data

The yeast population data is plotted in Figure II.12.

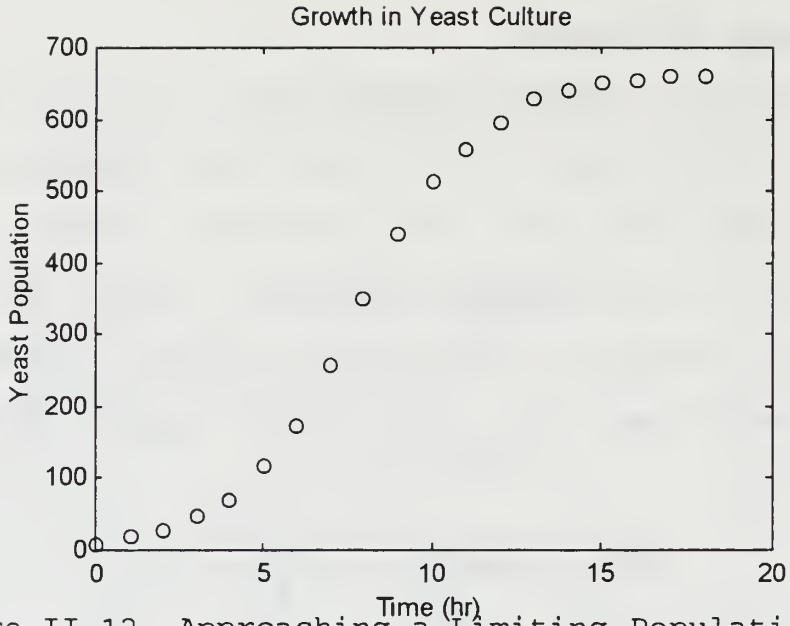

Figure II.12. Approaching a Limiting Population Level

Notice that the population does indeed tend toward some limiting value, which we estimate by inspection to be about 665. Now consider the model

$$\Delta p_n = p_{n+1} - p_n = k(665 - p_n)p_n \qquad (2.5)$$

which causes Δp_n to become increasingly small as p_n approaches 665. We can test the validity of this model by plotting $(p_{n+1} - p_n)$ versus $(665 - p_n)p_n$ to see if there is a reasonable proportionality, with slope k. This can be done with the following M-file:

```
pn=[9.6 18.3 29 47.2 71.1 119.1 174.6 257.3 350.7 441
513.3 559.7 594.8 629.4 640.8 651.1 655.9 659.6 661.8];
a=length(pn);
delta_pn=[];
for n=1:(a-1)
   delta_pn(n)=pn(n+1)-pn(n);
end
pn=pn(1:(a-1));
x=pn.*(665-pn);
```

```
k=rto(x,delta_pn);
plot(x,delta_pn,'o',x,k*x)
xlabel('pn(665-pn)')
ylabel('p(n+1)-p(n)')
title('Growth Constrained by Resources')
text(80000,62,['k = ',num2str(k)])
```

which produces the output

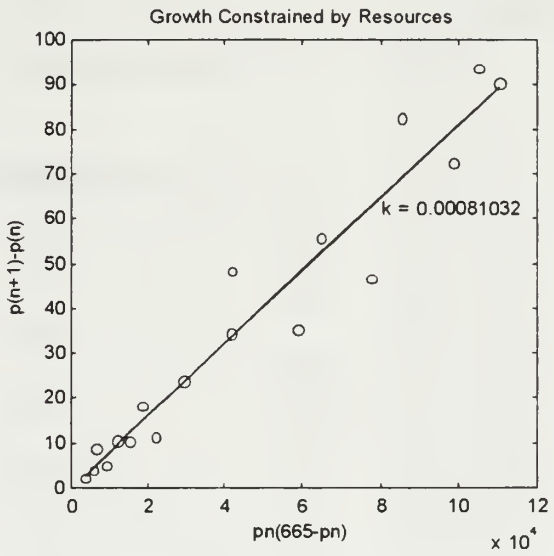

Figure II.13. Constrained Growth Model

4. Numerical Solution for the Refined Model

Using the value of k obtained above and solving for p_{n+1} yields the model

$$p_{n+1} = p_n + 0.00081(665 - p_n)p_n \tag{2.6}$$

which is quadratic in p_n. This model is easily solved numerically (iteratively) if we start with an initial population. Since p_0 = 9.6, we can compute p_1 as follows:

$$p_1 = p_0 + 0.00081(665 - p_0)p_0 = 9.6 + 0.00081(665 - 9.6)9.6 = 14.70$$

Next we can compute p_2 from p_1, and the remaining populations *predicted* by the model in similar fashion. Comparing these predictions with the actual observed populations gives insight into the accuracy of our model.

Time	Observation	Prediction
0	9.6	9.6000
1	18.3	14.6964
2	29.0	22.4377
3	47.2	34.1159
4	71.1	51.5497
5	119.1	77.1644
6	174.6	113.9060
7	257.3	164.7521
8	350.7	231.5098
9	441.0	312.7992
10	513.3	402.0354
11	559.7	487.6694
12	594.8	557.7172
13	629.4	606.1823
14	640.8	635.0622
15	651.1	650.4622
16	655.9	658.1218
17	659.6	661.7884
18	661.8	663.5100

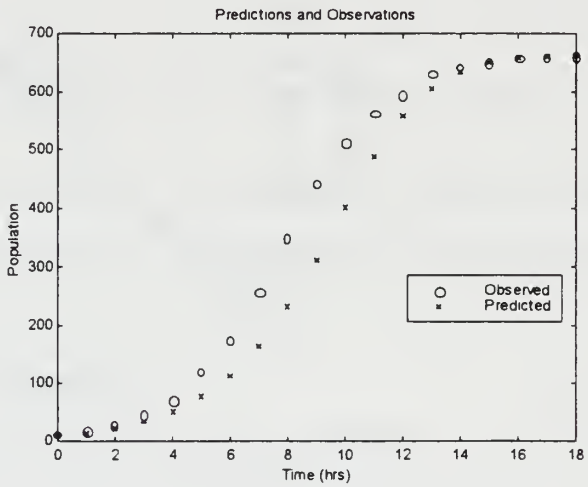

Figure II.14. Model Predictions and Observations

It is apparent that our model does a fairly good job of predicting population levels and captures the trend of the original data. Now let us study variations of this population model by changing the constant of proportionality k in Equation (2.5).

5. Bifurcation and Chaos

Suppose we have the population model

$$p_{n+1} = k(1 - p_n)p_n \tag{2.7}$$

where we have normalized the carrying capacity to **one** unit of population. We wish to examine the effect of changing the growth constant of proportionality, k. For this experiment, we will use an initial population p_0 = 0.3, evaluate p_{n+1} from Equation (2.7), and plot the population over time for various values of k.

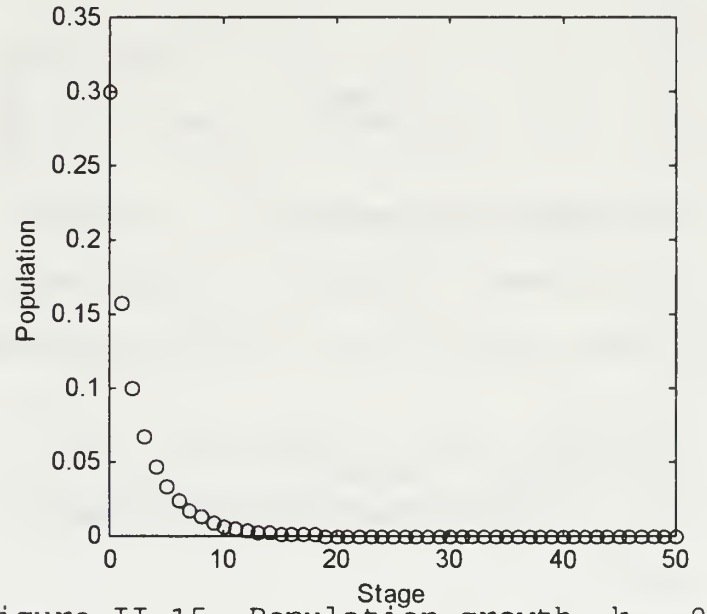

Figure II.15. Population growth, k = 0.75

We see in Figure II.15 that the population dies out quickly when k = 0.75. Figure II.16 shows the results of increasing k to 1.5.

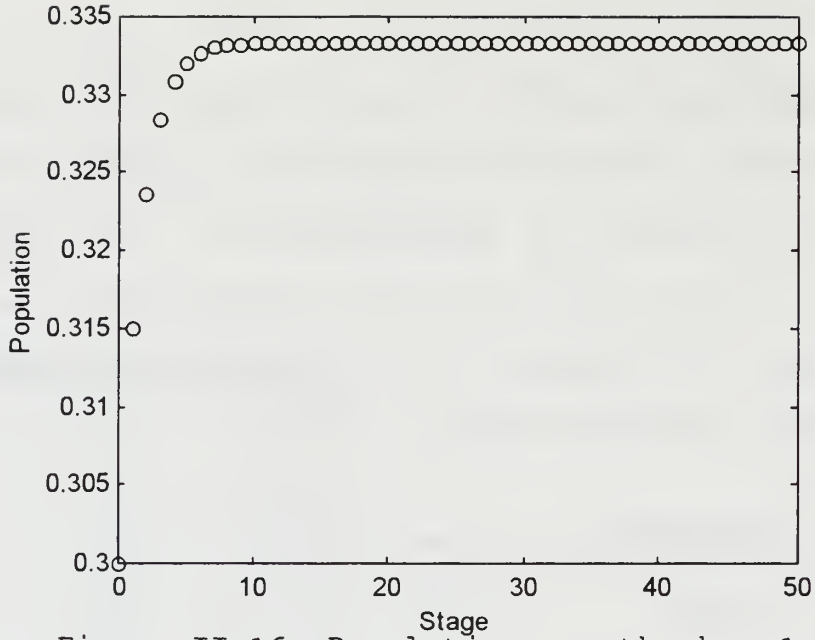

Figure II.16. Population growth, k = 1.5

Here, the larger proportionality constant causes the population to *increase* initially, then tend toward a constant value (about one-third) as the stage number grows.

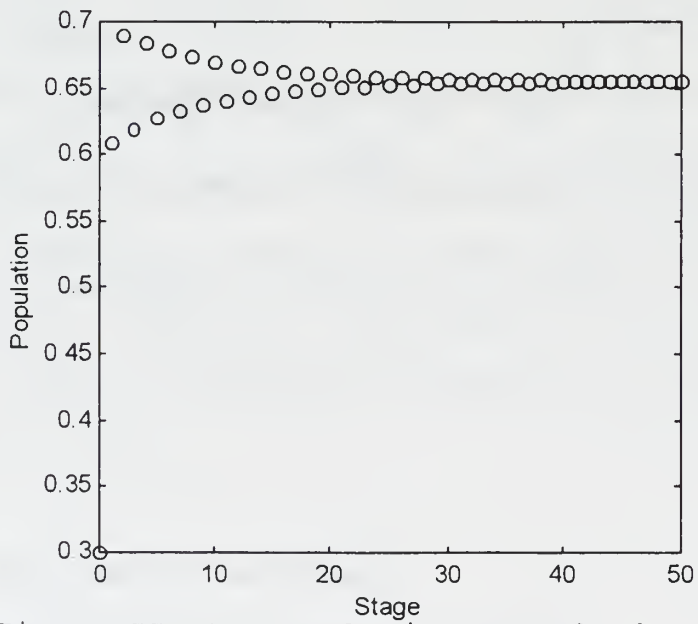

Figure II.17. Population growth, k = 2.9

Figure II.17 depicts the results of increasing k to 2.9. In this case, we begin to see *oscillation* in the early stages before the population gravitates toward a steady state of approximately 0.65. In Figure II.18, we increase the value of k to 3.25.

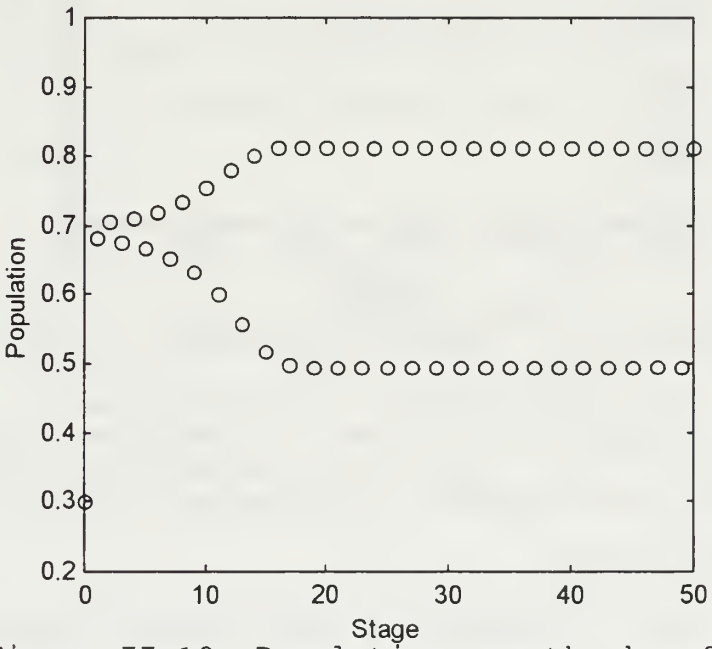

Figure II.18. Population growth, k = 3.25

For k = 3.25, we no longer see the population tend toward *one* steady-state value. In fact, as the stages increase, the population *splits* into two branches and *jumps* between values that tend toward steady-state values (approximately 0.5 and 0.8). This splitting effect is known as **bifurcation**.

Figure II.19 shows the affect of increasing k to 3.4.

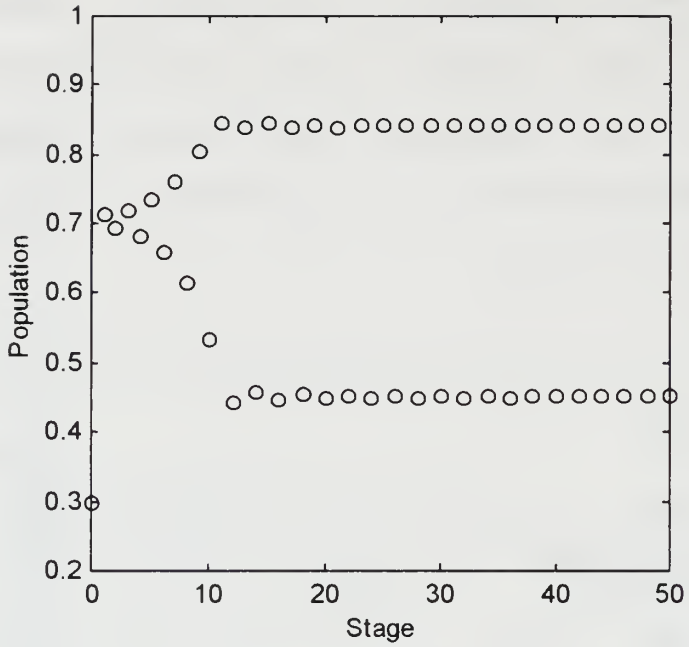

Figure II.19. Population growth, k = 3.4

We see in Figure II.19 that, in addition to splitting, each of the two branches oscillates before eventually achieving steady-state.

We find that increasing k even further causes each of the two branches to split, or bifurcate, again. Figure II.20 depicts this phenomenon.

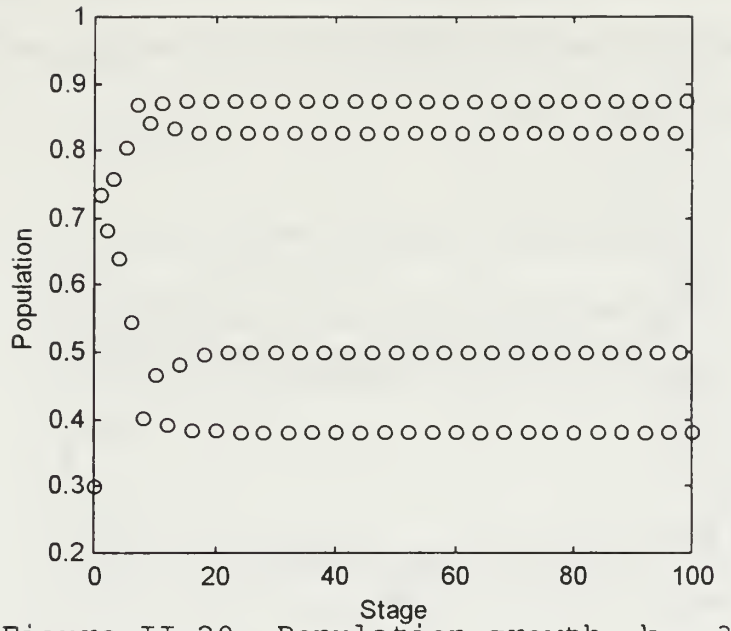

Figure II.20. Population growth, k = 3.5

Here, we see each branch bifurcating again. Let's look at the effect of increasing k again.

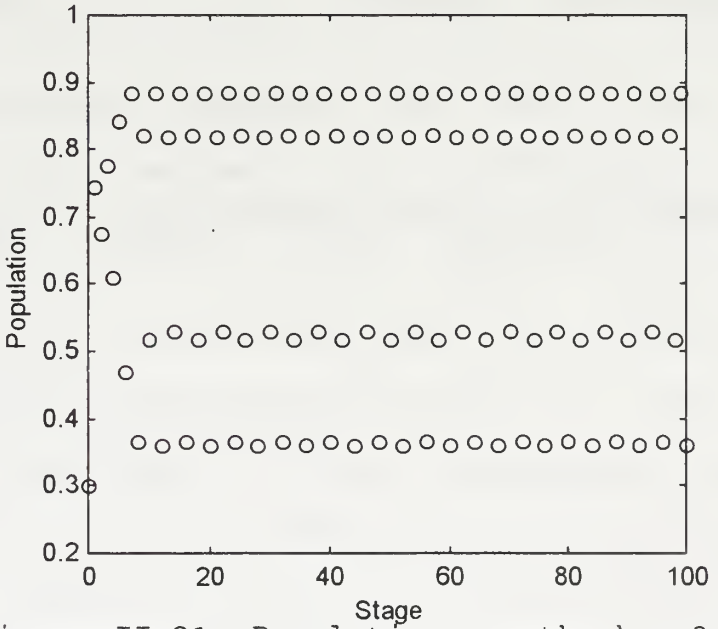

Figure II.21. Population growth, k = 3.545

Figure II.21 reveals that for k = 3.545, each of the four branches are oscillating initially before settling down and reaching steady-state.

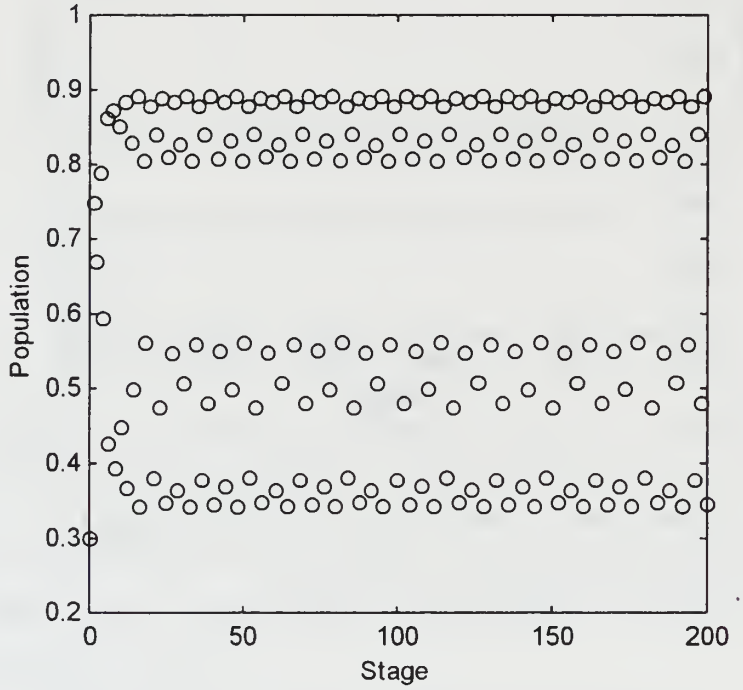

Figure II.22. Population growth, k = 3.695

Figure II.22 shows the results of increasing k to 5.695. We see each of the four branches shown in Figure II.21 begin to bifurcate again, forming eight total branches.

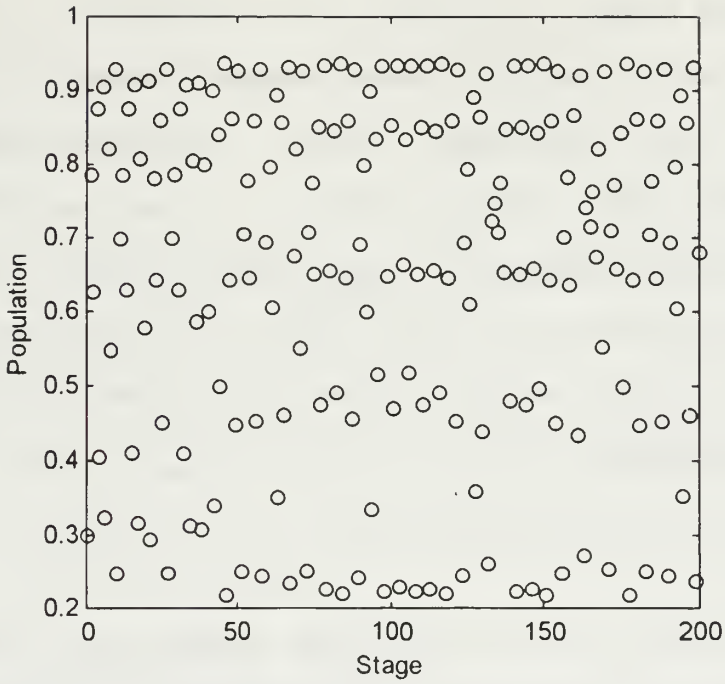

Figure II.9. Population growth, k = 3.75

Finally, we see in Figure II.9 that when k is large enough, the system reaches a point where there are **no** steady-state values. The system is trying to be at every value for every stage. At this point, the system has achieved a state of **chaos**.

D. EXAMPLE 3: LANCHESTER'S SQUARE LAW DISCRETE MODEL

Consider the situation of combat between two forces, which we will call Blue and Red. We want to know, among other things, if one side will defeat the other, or whether the fight will end in a draw. Lanchester's combat models provide an analytical framework within which to study these questions (see Ch. 11 of (2)). We now look at the discrete version of the Lanchester square law model.

1. The Model

Let B_t and R_t be the **force levels** (number of **systems** on each side; e.g., soldiers, tanks, or ships) of the Blue force and Red force, respectively, at time period t. If we assume that the **change** in force level (i.e., the *casualty rate*) for a given side (say, Blue) is proportional to the force level of the opposing side (Red), we can very easily derive the equation

$$B_{t+1} = B_t - aR_t, \tag{2.8}$$

where a is known as the attrition coefficient and reflects the rate at which the Red force inflicts casualties on the Blue force. A similar analysis results in the expression

$$R_{t+1} = R_t - bB_t, \tag{2.9}$$

where b is the coefficient reflecting, similarly, the rate at which Blue inflicts casualties on Red. Together, these equations form a discrete dynamical system with which we can predict and study the outcomes of hypothetical "battles."

2. A Numerical Example

Suppose that the Red and Blue forces engage in battle, and that the two sides begin the engagement with 50 and 100 systems, respectively (i.e., $R_0 = 50$ and $B_0 = 100$). Suppose also that 10 Red systems are required to destroy one Blue system (i.e., a = 0.1) and furthermore that 20 Blue systems

are required to destroy one Red system (b = 0.05). Which side will win this battle? To find out, we could execute the following M-file:

```
% Initialize force vectors, initial force strengths
% and attrition coefficients:

B=[]; R=[]; B(1)=100; R(1)=50;
a=.1; b=.05;

% Loop through Lanchester equations to determine
% force strength at each time period:

for t=1:12  %12 is an arbitrary stopping point
   B(t+1)=B(t)-a*R(t);
   R(t+1)=R(t)-b*B(t);
end
t=0:t;
% create a matrix of output
OUT=[t' B' R' (B./R)'];
% write this matrix to an ASCII file for future
% reference:
dlmwrite('f:\thesis\documents\lanch.out',OUT,'\t');
```

Notice the use of the percent sign (%). This symbol denotes comments in an M-file which are not executed. These comments can appear anywhere in the file, as long as a percent sign precedes them. Notice also the command **dlmwrite**, which saves the results of M-file execution in an ASCII file (to later be manipulated with a word processor, spreadsheet, or other software). With little effort, the file "lanch.out" created by MATLAB can be imported into this document and displayed as a table:

t	B_t	R_t	B_t/R_t
0	100	50	2
1	95	45	2.1111
2	90.5	40.25	2.2484
3	86.475	35.725	2.4206
4	82.9025	31.4013	2.6401
5	79.7624	27.2561	2.9264
6	77.0368	23.268	3.3108
7	74.71	19.4162	3.8478
8	72.7683	15.6807	4.6406
9	71.2003	12.0423	5.9125
10	69.9961	8.4822	8.2521
11	69.1478	4.9824	13.8783
12	68.6496	1.525	45.0148

Table II.4. Status of Red and Blue Forces (a=.1, b=.05)

Blue requires only 12 time periods to virtually eliminate the Red force, even though the Red force systems were twice as lethal (a = 0.1 vs. b = 0.05). Apparently, a 2:1 lethality advantage (ratio of attrition coefficients) does not make up for a 2:1 disadvantage in initial force levels. Let's run the model again using a = 0.2 to see if an increase in lethality for the Red force achieves better results (see Table II.5).

t	B_t	R_t	B_t/R_t
0	100	50	2
1	90	45	2
2	81	40.5	2
3	72.9	36.45	2
4	65.61	32.805	2
5	59.049	29.5245	2
6	53.1441	26.5721	2
7	47.8297	23.9148	2
8	43.0467	21.5234	2
9	38.742	19.371	2
10	34.8678	17.4339	2
11	31.3811	15.6905	2
12	28.243	14.1215	2

Table II.5. Status of Red and Blue Forces (a=.2, b=.05)

In this case, the original 2:1 force ratio (B_t/R_t) is maintained at every time period, indicating that **parity** exists. Clearly, Red requires a higher lethality advantage to compensate for the 2:1 disadvantage in initial force strength. The lethality advantage required is 4/1, which is the **square** of the 2/1 force strength disadvantage for Red. This is no coincidence. One of the results Lanchester derived from (2.8) and (2.9) is an expression that must be satisfied at every time period for parity to occur:

$$bB^2 = aR^2 \tag{2.10}$$

Given that b = .05, B = 100, and R = 50, we can use this expression to compute the required value of a in order to achieve parity:

$$0.05(100)^2 = a(50)^2 \Rightarrow a = \frac{0.05(100)^2}{50^2} = 0.2$$

Table II.5 verifies this to be true. In order for the Red force to win, we must have a > 0.2. To demonstrate this, Table II.6 displays the results of the battle for b = 0.05, B_0 = 100, R_0 = 50, and a = 0.3.

t	B_t	R_t	B_t/R_t
0	100	50	2
1	85	45	1.8889
2	71.5	40.75	1.7546
3	59.275	37.175	1.5945
4	48.1225	34.2113	1.4066
5	37.8591	31.8051	1.1903
6	28.3176	29.9122	0.94669
7	19.3439	28.4963	0.67882
8	10.7951	27.5291	0.39213
9	2.5363	26.9893	0.093975

Table II.6. Status of Red and Blue Forces (a=0.3, b=.05)

III. MODELING USING PROPORTIONALITY

In Chapter II, the concept of proportionality was introduced along with a demonstration of its use in modeling change (see Example 2). This concept is now formalized by applying MATLAB to two examples taken from Chapter 4 of (2).

A. EXAMPLE 1: VEHICULAR STOPPING DISTANCE

This problem, first introduced in Chapter 2 and presented in more depth in Chapter 4 of (2), asks the modeler to predict a vehicle's total stopping distance as a function of its speed.

1. Initial Model

Consider the following rule of thumb often provided to young drivers-in-training: for every 10 miles per hour (mph) of your speed, leave one car length (about 15 feet) between you and the vehicle in front of you. For example, a car traveling at 60 mph should move no closer than 80 feet from the vehicle in front. This rule assumes a **linear** relationship between stopping distance and velocity. A graph of this proportionality relationship is a straight line of positive slope passing through the origin, as depicted in Figure III.1.

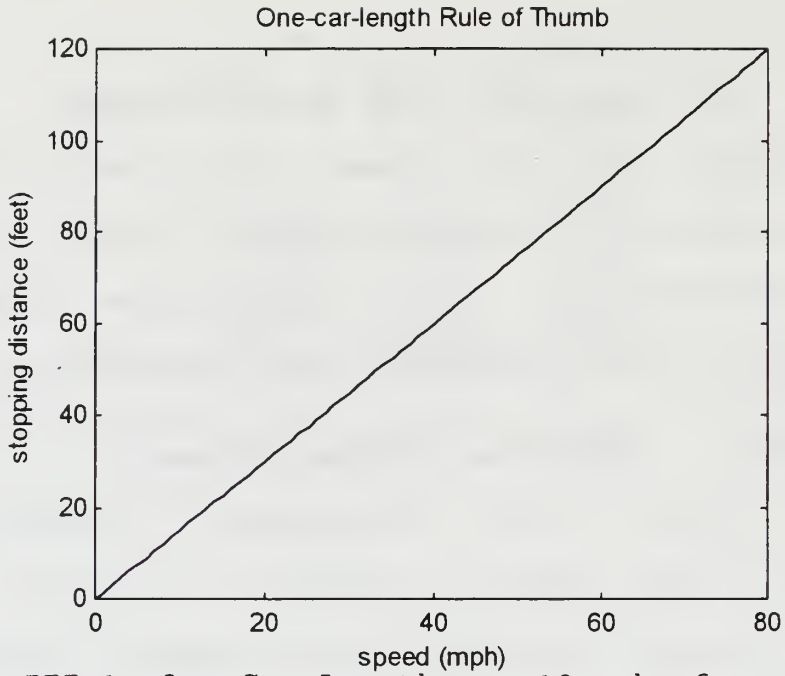

Figure III.1. One Car Length per 10 mph of speed is a proportionality relationship.

To verify this potential model, we need to test it against empirical data. We'll use the following data taken from Chapter 4 of (2):

Speed (mph)	Driver Reaction Distance (ft)	Braking Distance (ft)		Total Stopping Distance (ft)	
20	22	18-22	(20)	40-44	(42)
25	28	25-31	(28)	53-59	(56)
30	33	36-45	(40.5)	69-78	(73.5)
35	39	47-58	(52.5)	86-97	(91.5)
40	44	64-80	(72)	108-124	(116)
45	50	82-103	(92.5)	132-153	(142.5)
50	55	105-131	(118)	160-186	(173)
55	61	132-165	(148.5)	193-226	(209.5)
60	66	162-202	(182)	228-268	(248)
65	72	196-245	(220.5)	268-317	(292.5)
70	77	237-295	(266)	314-372	(343)
75	83	283-353	(318)	366-436	(401)
80	88	334-418	(376)	422-506	(464)

Table III.1. Observed Reaction and Braking Distances (Mean Distances in Parentheses)

Let's compare a scatterplot of these data against the One-car-length rule proportionality (we'll use the Mean Total Stopping Distance data from Table III.1 for this comparison). This comparison is depicted in Figure III.2.

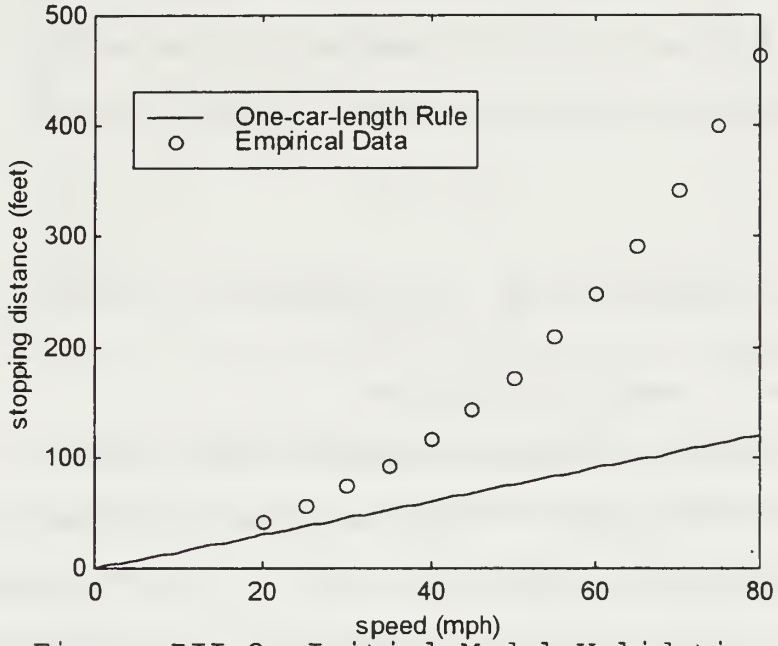

Figure III.2. Initial Model Validation

From the plot, it appears that the rule of thumb does not accurately predict safe stopping distances, especially at speeds above 40 mph. It stands to reason that the model must be refined to make it more realistic. Notice that the scatterplot of the data is more accurately represented by a curve rather than a straight line through the origin.

2. Model Refinement

In order to construct a better model, let us consider the components that make up the total stopping distance.

One can reasonably argue that the total distance d required for a vehicle to come to a complete stop is the sum of two distances: **reaction distance,** d_r (the distance traveled between the time the driver determines the need to stop and the time the driver applies the brakes), and **breaking distance,** d_b (the distance traveled from between application of the brakes and complete stop). In other words,

$$d = d_r + d_b \tag{3.1}$$

We now examine these two components individually.

a) Reaction Distance

Since we desire a model that predicts stopping distance as a function of vehicle speed, it seems natural to seek a relationship between *each of the components* in Equation (3.1) and vehicular speed. To get some idea of the relationship between reaction distance and speed, first create a scatterplot of the reaction distance data from Table III.1. This scatterplot is depicted in Figure III.3.

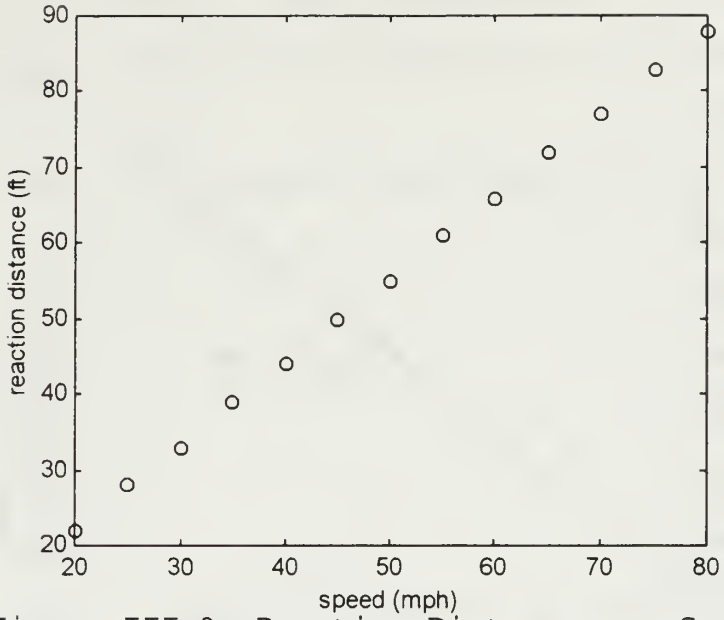

Figure III.3. Reaction Distance vs. Speed

This relationship appears to be linear, but is it in fact a proportionality? In order for reaction distance to be proportional to vehicular speed, the best-fit line must pass (reasonably) through the origin. The following M-file generates a plot to help in answering that question:

```
dr=[22 28 33 39 44 50 55 61 66 72 77 83 88];
speed=20:5:80;
tr=rto(speed,dr);
x=0:80;
plot(speed,dr,'o',x,tr*x)
text(40,40,['Slope = ',num2str(tr)])
xlabel('speed (mph)')
ylabel('reaction distance (ft)')
```

The plot is shown in Figure III.4.

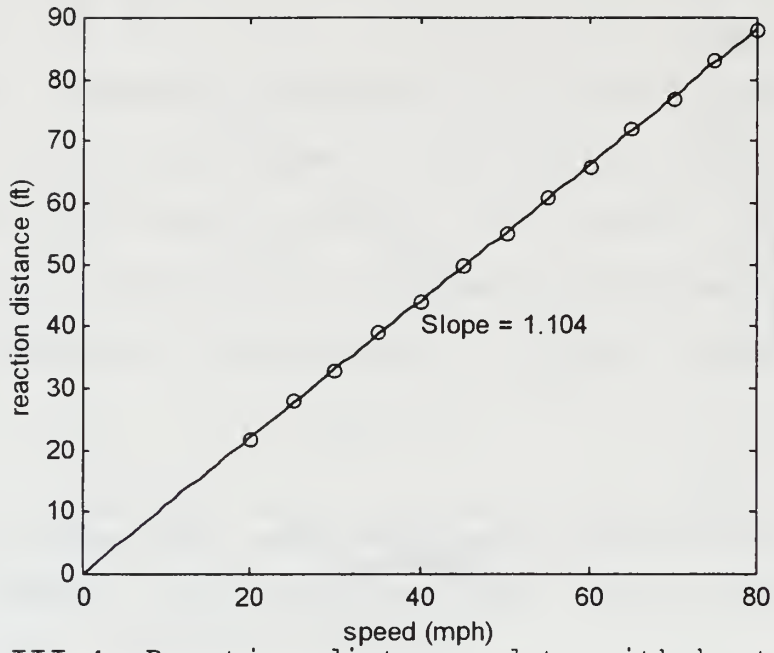

Figure III.4. Reaction distance data with best-fit line

Indeed, from the graph it can be seen that reaction distance is proportional to vehicle speed. The relationship is described by the submodel

$$d_r = 1.104v, \tag{3.2}$$

where v is the speed of the vehicle. Let us next determine a relationship between braking distance and vehicular velocity.

b) Braking Distance

To obtain some idea of the relationship between braking distance and vehicle speed, create a scatterplot of the braking distance data (we use the *mean* data for braking

distance, as before). The scatterplot is displayed in Figure III.5.

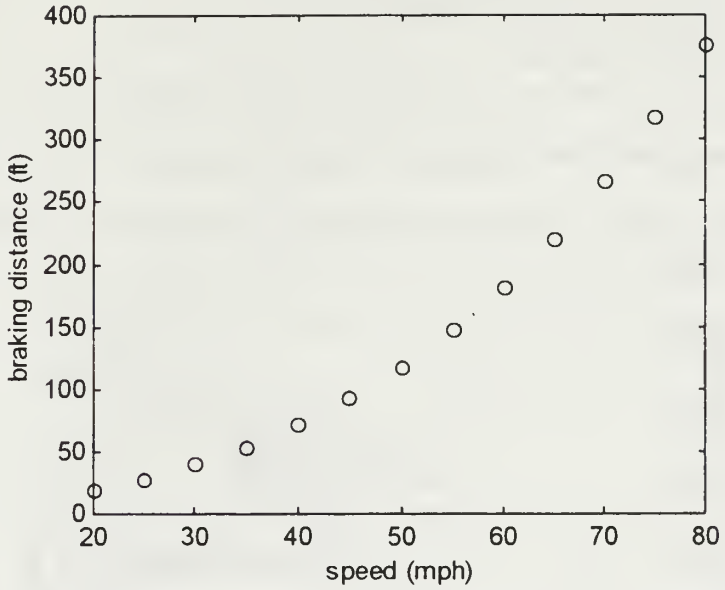

Figure III.5. Braking Distance vs. Speed

From the figure, it is quickly revealed that this relationship is definitely not linear, much less a proportionality. Since Braking Distance is not proportional to speed, it could turn out to be proportional to some **transformation** of speed. In other words, a plot of Braking Distance against some *function* of the speed (i.e., square root, square, etc.) could reasonably produce a straight line through the origin. As discussed in Section 4.2 of (2), it is reasonable to expect the braking distance to be proportional to the square of the speed. Let us try plotting Braking Distance vs. Speed^2. To do so, us the following M-file:

```
db=[20 28 40.5 52.5 72 92.5 118 148.5 182 220.5 266 318 376];
speed=20:5:80;
speed2=speed.^2;
k=rto(speed2,db);
x=0:max(speed2);
plot(speed2,db,'ko', x,x*k)
text(4000,200,['Slope = ',num2str(k)])
xlabel('speed^2 (mph^2)')
ylabel('braking distance (ft)')
```

The result is shown in Figure III.6.

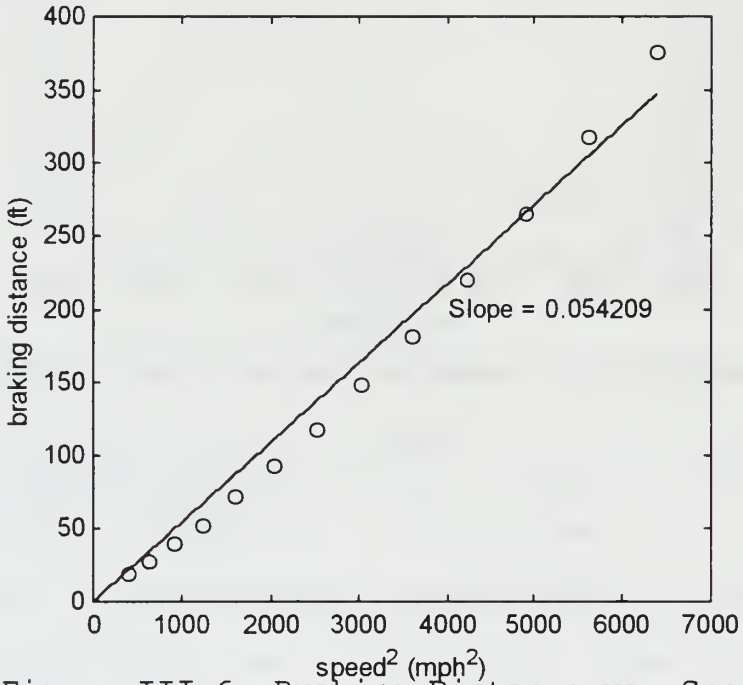

Figure III.6. Braking Distance vs. Speed^2

As you can see, this relationship appears reasonably close to a proportionality. From this we can tentatively conclude (pending model verification) that

$$d_b = 0.0542v^2. \qquad (3.3)$$

Summing the two submodels (3.2) and (3.3) yields the following proposed model for the total stopping distance:

$$d = 1.104v + 0.0542v^2 \qquad (3.4)$$

The predictions given by model (3.4) along with the actual observations are compared in Figure III.7. The One-car-length Rule is also plotted for comparison.

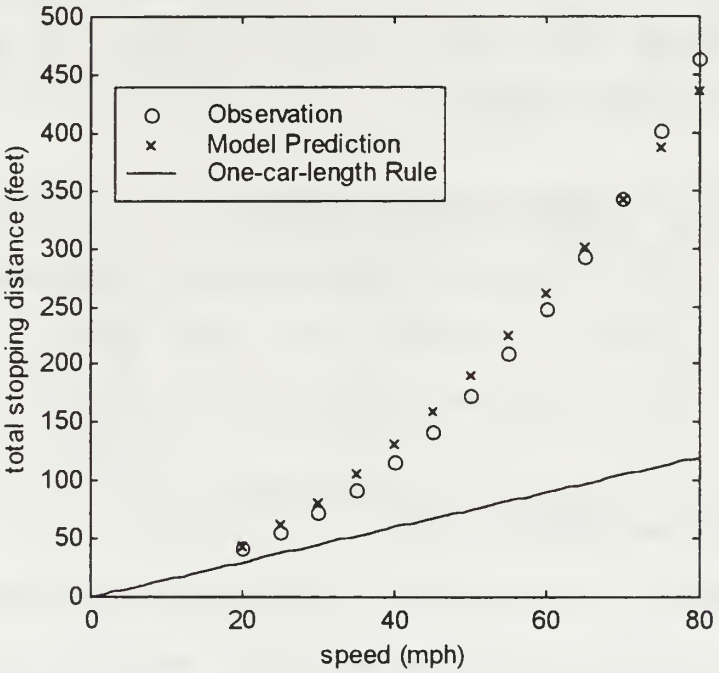

Figure III.7. Total Stopping Distance

The M-file used to produce Figure III.7 appears below:

```
d=[42 56 73.5 91.5 116 142.5 173 209.5 248 292.5 343 401 464];
speed=20:5:80;
x=0:80;
stop_d=1.104*speed+.0542*speed.^2;
plot(speed,d,'o',speed,stop_d,'x',x,1.5*x)
xlabel('speed (mph)')
ylabel('total stopping distance (ft)')
legend('Observation','Model Prediction','One-car-length Rule')
```

Considering the assumptions and simplifications necessary to construct model (3.4) together with the inherent inaccuracies in the data, our model appears to agree fairly closely with the data. Observe also that the One-car-length Rule significantly underestimates the required stopping distance at speeds above 40 mph.

This example demonstrates the use of proportionality models using transformed data. Let's consider another way that proportionality can be used to model a real-world situation.

B. EXAMPLE 2: A BASS FISHING DERBY

Consider the problem of determining (approximately) the weight of a fish in terms of some easily measurable dimension(s) (as discussed in Section 4.5 of (2)).

1. Initial Model

For simplicity's sake (at least initially), we restrict our analysis to one species of fish, say bass. Assuming that all bass are **geometrically similar** (see Section 4.4 of (2) for a discussion of geometric similarity), one can easily argue that the volume of any bass is proportional to the cube of some characteristic dimension. Using length l as our characteristic dimension, the volume V of a bass then satisfies the proportionality

$$V \propto l^3.$$

Suppose too that the average density of all bass is constant (not an unreasonable assumption since their bones are small and almost "fleshy"). Then, since weight W is volume times average density times gravity, it follows immediately that

$$W \propto l^3.$$

Let's compare this model against the following data, collected during a fishing derby (see Section 4.5 of (2)):

Length, l (in.)	Girth, g (in.)	Weight, W (oz.)
14.5	9.75	27
12.5	8.375	17
17.25	11.0	41
14.5	9.75	26
12.625	8.5	17
17.75	12.5	49
14.125	9.0	23
12.625	8.5	16

Table III.2. Observed Data

If our model is to be valid, the graph of W vs. l^3 should approximate a straight line passing through the origin. This plot, together with a best-fitting straight line is presented in Figure III.8. The following M-file generated this plot:

```
length=[14.5 12.5 17.25 14.5 12.625 17.75 14.125 12.625];
wt=[27 17 41 26 17 49 23 16];
l3=length.^3;
k=rto(l3, wt);
x=0:max(l3);
plot(l3,wt,'o',x,k*x)
xlabel('length^3 (in^3)')
```

```
ylabel('weight (oz.)')
text(1500,10,['slope = ',num2str(k)])
```

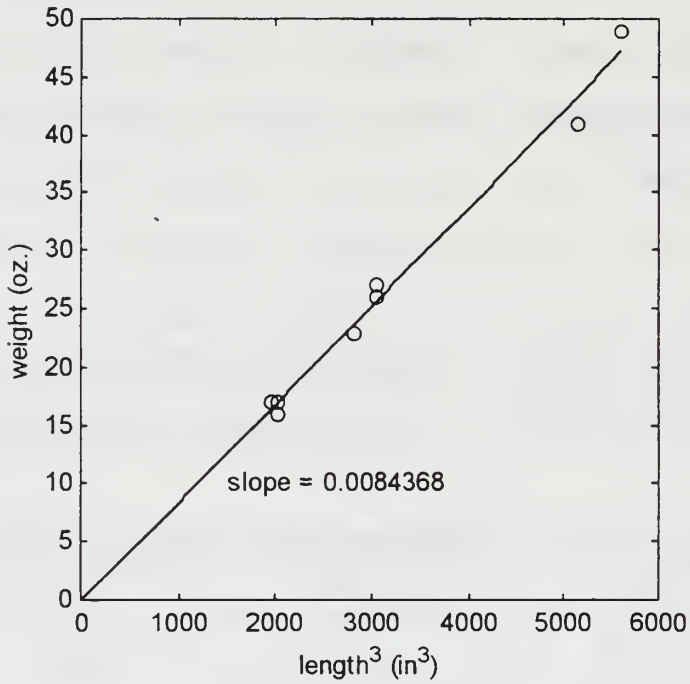

Figure III.8. Weight vs. length^3 with Best-fit Line

The slope of the best-fit line yields the proportionality factor k. With this value, we propose the model

$$W = 0.008437l^3 \qquad (3.5)$$

Figure III.9 compares model (3.5) with a scatterplot of the original data. This figure was generated by the following M-file:

```
length=[14.5 12.5 17.25 14.5 12.625 17.75 14.125 12.625];
wt=[27 17 41 26 17 49 23 16];
x=10:.1:20;
pred_w=0.008437*x.^3;
```

```
plot(length,wt,'o',x,pred_w)
xlabel('length (in)')
ylabel('weight (oz.)')
legend('Original data','Model (3.5)')
```

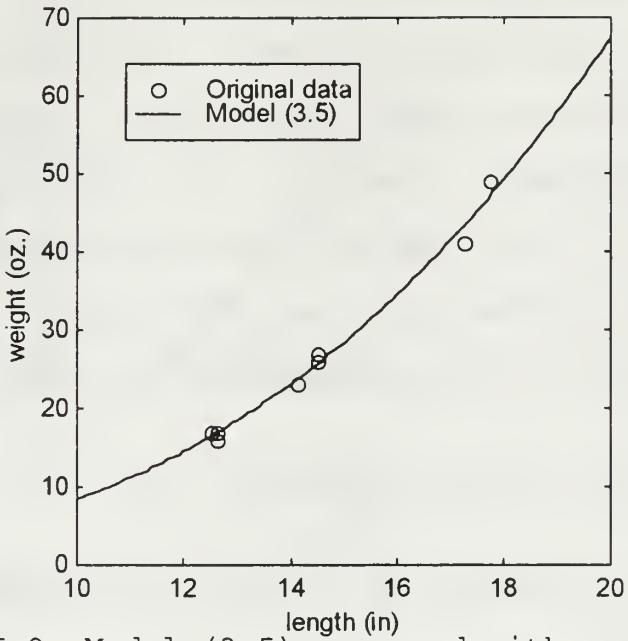

Figure III.9. Model (3.5) compared with original data

With the small amount of data we have, Figure III.9 shows that model (3.5) does not appear grossly inaccurate. However, suppose certain fishermen are dissatisfied with this model because it treats long, skinny fish equally with short, fat fish (that is, our model, which is based solely on length, could predict a long, skinny fish to weigh more than a short, fat fish; not very satisfactory for a fisherman). Let us propose an alternate model to satisfy these disgruntled fishermen.

2. Model Refinement

To take into account the three-dimensional aspect of a given fish, we replace the assumption that all fish are geometrically similar by assuming that only the *cross-sectional areas* of the fish are similar. Choosing *girth g* as our characteristic dimension (we define girth as the circumference of the fish at its widest point), then the average cross-sectional area is proportional to the square of the girth, and we can subsequently conclude that

$$W \propto \lg^2$$

(see Section 4.5 of (2) for a detailed discussion of this proportionality).

Let's attempt to verify this model using the data from Table III.2. First we create the following M-file:

```
length=[14.5 12.5 17.25 14.5 12.625 17.75 14.125 12.625];
wt=[27 17 41 26 17 49 23 16];
girth=[9.75 8.375 11 9.75 8.5 12.5 9 8.5];
g2=girth.^2;
lg2=length.*g2;
k=rto(lg2, wt);
x=0:max(lg2);
plot(lg2,wt,'o',x,k*x)
xlabel('lg^2 (in^3)')
ylabel('weight (oz.)')
text(600,10,['slope = ',num2str(k)])
```

The M-file produces the graph and best-fitting line displayed in Figure III.10.

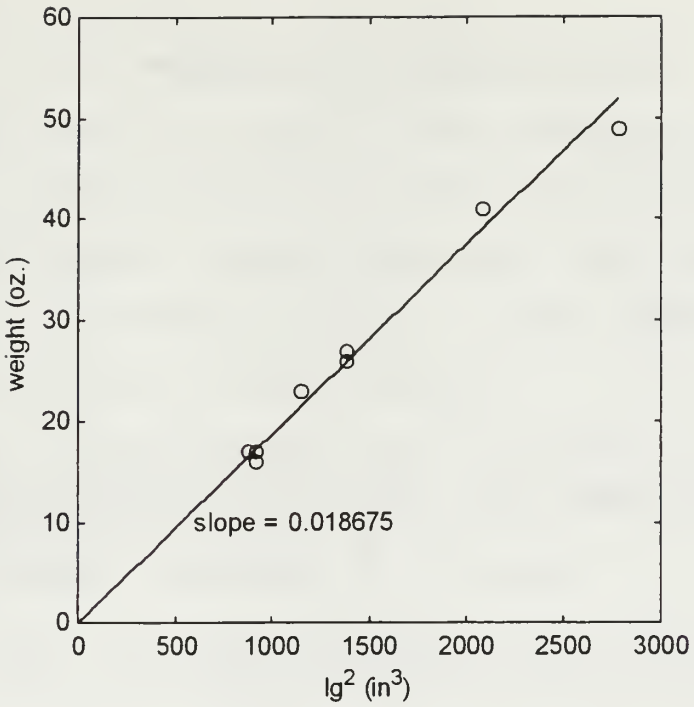

Figure III.10. Wt vs. lg^2 w/Best-fit Line

The result of this procedure yields the model

$$W = 0.0187 \mathrm{lg}^2. \qquad (3.6)$$

A fisher would probably be happier with the new model (3.6), because doubling the girth leads to a fourfold increase in the predicted weight of the fish. However, this rule is more difficult to apply. It requires two measurements for each fish as opposed to only one for model (3.5).

IV. MODELING FROM DISCRETE DATA

In this chapter, we look at two techniques for modeling data sets: Model Fitting and Empirical Modeling. The distinctions between these techniques are best seen by defining the following possible tasks associated with analyzing a collection of data points:

- Fitting a selected model type (or types) to the data
- Choosing an appropriate model from competing ones determined by the data
- Making predictions based on the model

The first two tasks describe the situation where a model type (e.g., curve) *exists* that seems to explain the behavior being observed (e.g., a quadratic explaining projectile motion). The modeler's job is to find the particular model that "best" fits the data. If more than one such model is found, he must choose, using some established criteria, the best of the alternatives to describe the phenomenon being studied. This process is called *Model Fitting*.

For the third task, a model type *does not exist* to explain the observed behavior. Yet, the modeler wishes to *predict* what might happen within a certain range of interest (either within or outside the range of the data) based solely on the observed data. This process is known as *Empirical Modeling*.

In discussing these different modeling techniques, several new MATLAB capabilities will be introduced. Among these are **polyfit**, which fits a polynomial of a desired order to a given set of data points; and **spline**, which fits a cubic spline to a set of data points.

A. MODEL FITTING

The idea of choosing a model thought to describe some observed behavior is not new to us. The Vehicular Stopping Distance problem is one such example. Recall that, after decomposing the problem into two sub-models (reaction distance and braking distance) we proceeded to fit submodels to each component. We had reason to believe (even before seeing the data) that reaction distance was proportional to vehicle speed based on the premise that, regardless of speed, response time was essentially constant (an average response time across the population). Likewise, we deduced from the physics of the vehicle braking process that braking distance was proportional to the speed squared. Once these proportionalities were established, only the constants of proportionality remained to be determined. In both cases, the observed data also validated the reasonableness of the submodels chosen. The two combined submodels then provided a relatively accurate predictor of total required stopping distance as a function of vehicle speed.

1. Vehicular Stopping Distance -- Another Approach

Recall that in determining the constant k for the expression $d_b = kv^2$ we first transformed (by squaring) the vehicular speed data, and then plotted braking distance versus speed squared. The value of the slope of the best-fit line (using the least-squares criterion) through the origin then yielded the proportionality constant k, resulting in model (3.3). An alternative approach would be to compute this constant directly, without first executing any transformations. The following M-file demonstrates this approach (the plot generated by this M-file appears at Figure IV.1):

```
db=[20 28 40.5 52.5 72 92.5 118 148.5 182 220.5 266 318 376];
speed=20:5:80;
k=cfit(speed,db,2);
plot(speed,db,'ko',speed,k*speed.^2)
legend('observed data',['model: db = ',num2str(k),'v^2'])
xlabel('speed, v (mph)')
ylabel('braking distance, db (ft)')
```

The function **k = cfit(x,y,n)** computes the constant k (in the least squares sense) for the one-term polynomial expression $y = kx^n$. Like the function **rto(x,y)**, this function is not available in MATLAB but can be found in the appendix.

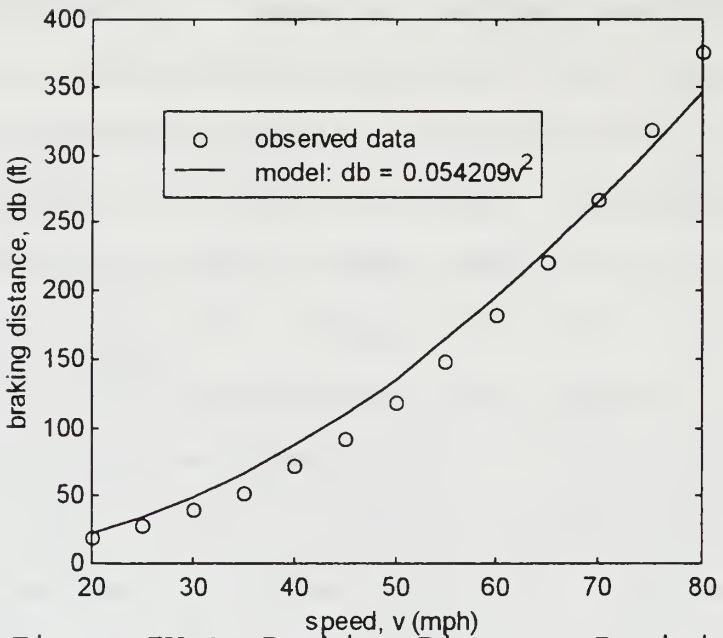

Figure IV.1. Braking Distance Revisited

Notice that the resulting model obtained using the function **cfit** is very close to the model shown in Figure III.6 obtained using the function **rto**. This is another case where similar results can be obtained using different techniques with MATLAB.

Because it can generate one-term polynomial models of the form $y = kx^n$ for any degree n (including fractional degrees), the function **cfit** is useful in many model-fitting scenarios.

2. Residual Plots

A useful tool in determining the validity of a model is the study of the errors or *residuals* (the differences

between predicted and observed values). We can compute and plot these residuals using the following M-file:

```
db=[20 28 40.5 52.5 72 92.5 118 148.5 182 220.5 266 318 376];
speed=20:5:80;
residual=.054209*speed.^2-db;
plot(speed,residual,'ko',speed,zeros(size(speed)))
xlabel('speed, v (mph)')
ylabel('residual (ft)')
```

Figure IV.2 shows the resulting residual plot.

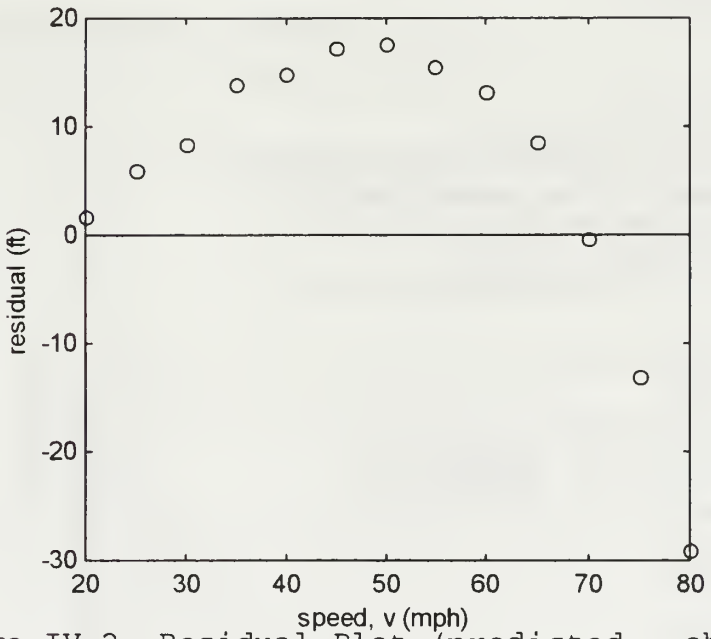

Figure IV.2. Residual Plot (predicted - observed)

Note the distinct pattern in the nature of the residuals. This pattern might cause us to reexamine the model for erroneous or oversimplified assumptions, or other errors. For this particular case, however, the magnitude of the errors is almost insignificant (about one car length or less), and further investigation is unwarranted.

3. Using MATLAB's Polyfit Function

MATLAB has a built-in function for fitting polynomials to a data set called polyfit. The function **polyfit(x,y,n)** produces the n+1 coefficients of the n^{th} degree polynomial that fits (in the least squares sense) the data depicted by the vectors x and y. The M-file below, which produces Figure IV.3, shows how this function can be used to model braking distance:

```
db=[20 28 40.5 52.5 72 92.5 118 148.5 182 220.5 266 318 376];
speed=20:5:80;
p=polyfit(speed,db,2);
predict=p(1)*speed.^2+p(2)*speed+p(3);
plot(speed,db,'ko',speed,predict)
legend('observed data','prediction')
xlabel('speed, v (mph)')
ylabel('braking distance, db (ft)')
```

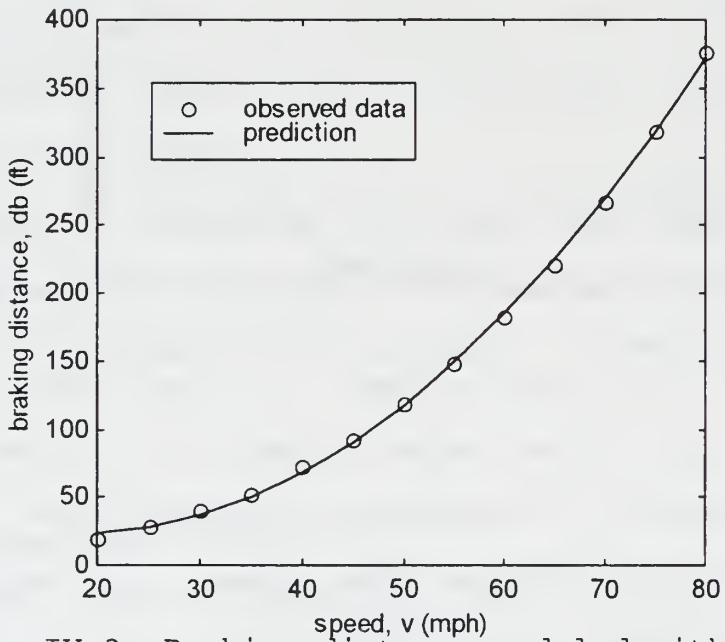

Figure IV.3. Braking distance modeled with polyfit

The equation for the model curve in Figure IV.3 generated by polyfit is

$$d_b = 0.0887v^2 - 3.0841v + 50.1294 \qquad (4.1)$$

Notice how this curve appears to more accurately capture the data than the model having only the squared term (model (3.3), plotted in Figure IV.1). The residual plot for model (4.1) provides some verification of this apparent increase in accuracy (Figure IV.4).

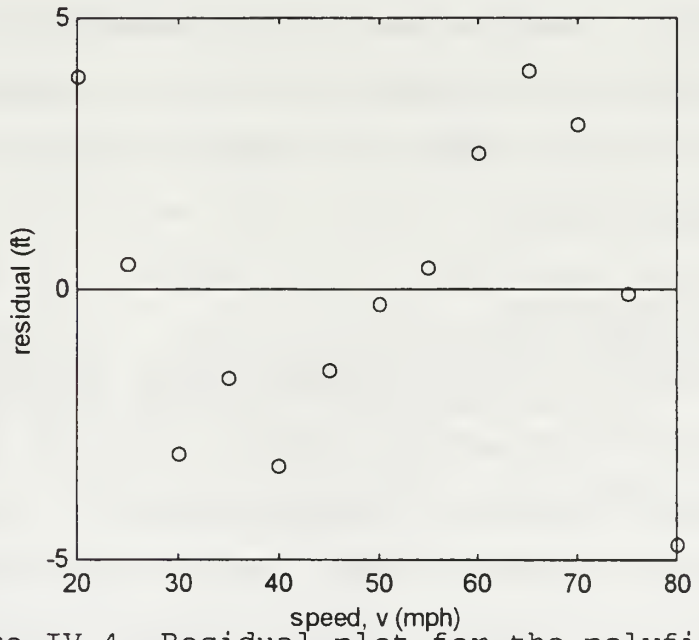

Figure IV.4. Residual plot for the polyfit model

Compare these residuals with those shown in Figure IV.1. Figure IV.4 seems to provide further evidence that model (4.1) is more accurate than (3.3), but is this actually the case? Upon closer inspection, we see that

model (4.1) predicts that a vehicle traveling at a speed of zero mph requires over 50 feet to come to a complete stop. This absurdity reveals the potential risk of blindly using this function (or any other, for that matter). Polyfit produces a polynomial that fits the data very well; it may well serve to allow the modeler to interpolate between data points. However, one would be ill-advised to use this model (4.1) to attempt to extrapolate information about vehicular braking distances for speeds outside the range of the observed data.

The previous example does not necessarily point out a weakness in MATLAB, but it does warn the user against merely using polyfit to fit data to a curve without completing further model analysis.

B. EMPIRICAL MODELING

In the braking distance example above, physics dictates that the relationship between braking distance and vehicle speed is a proportionality to speed squared. Many modeling scenarios arise, however, where neither physics nor any other science can provide a formula modeling the behavior or phenomenon being studied. In these situations, we typically have only the data itself from which to make desired predictions (either within or outside the range of the data). Thus, we wish to find some curve of convenience, or

empirical model, that captures the trend of the data from which to best make our predictions.

In this section, we discuss several types of empirical models, including high- and low-order polynomials, and cubic splines. A detailed discussion of these concepts, along with other topics related to empirical modeling, can be found in Chapter 6 of (2).

1. High-Order Polynomial Models

Mathematical theory from linear algebra guarantees that a unique polynomial of at most degree n can be passed exactly through a set of n+1 distinct (x,y) points (see p. 180 of (2)). So, for any set of data we wish to analyze, we can find a polynomial that fits it *exactly*, with no error. As good as this sounds, we'll find that this method has several drawbacks. Let us use MATLAB's **polyfit** function to compute these polynomials in several examples.

a) The Elapsed Time of a Tape Recorder

The data below relates the counter on a particular analog tape recorder to its elapsed playing time (*c* represents counter reading and *t* the elapsed time in seconds):

c	100	200	300	400	500	600	700	800
t	205	430	677	945	1233	1542	1872	2224

The following M-file computes and plots the 7^{th}-order polynomial that passes exactly through each data point:

```
c=100:100:800;
t=[205 430 677 945 1233 1542 1872 2224];
p=polyfit(c,t,7);
x=0:800;
y=polyval(p,x);
plot(c,t,'o',x,y)
xlabel('counter reading')
ylabel('elapsed time (sec)')
legend('observed data','model prediction')
```

Note the use of the function **polyval(p,x)** in this file. Polyval evaluates the polynomial generated by **polyfit** at each point of the vector *x*. The input argument *p* is the vector of coefficients of the polynomial.

The coefficients of the 7^{th}-degree polynomial generated by this M-file are:

$$
\begin{aligned}
a_0 &= -1.4000e+001 \\
a_1 &= \ \ 2.3291e+000 \\
a_2 &= -2.9083e-003 \\
a_3 &= \ \ 1.9785e-005 \\
a_4 &= -5.3542e-008 \\
a_5 &= \ \ 8.0139e-011 \\
a_6 &= -6.2500e-014 \\
a_7 &= \ \ 1.9841e-017
\end{aligned}
$$

Figure IV.5 shows a scatterplot of the data along with the polynomial model. It is evident that the polynomial model passes through each of our data points. In addition, it appears that the model captures the trend of the data fairly well for regions between data points, as

well as outside the range of the data. Overall, this 7^{th}-degree polynomial model seems to be a fairly good model.

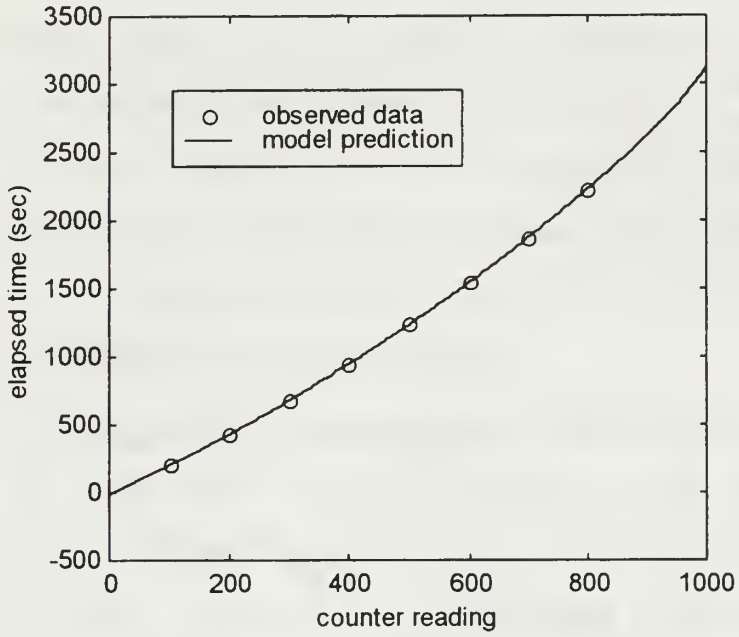

Figure IV.5. Tape recorder counter data with high-order polynomial model

Let us look at another example using high-order polynomials.

b) Volume of a Ponderosa Pine

In this problem, we wish to predict the volume V (in board feet) of Ponderosa Pine trees based on the tree's diameter d. The following data represents a sample of 14 such trees (taken from p. 187 of (2)):

d	17	19	20	22	23	25	31	32	33	36	37	38	39	41
V	19	25	32	51	57	71	141	123	187	192	205	252	248	294

The following M-file computes the 13^{th}-degree polynomial model and plot in Figure IV.6:

```
d=[17 19 20 22 23 25 31 32 33 36 37 38 39 41];
v=[19 25 32 51 57 71 141 123 187 192 205 252 248 294];
p=polyfit(d,v,13);
x=17:.1:41;
y=polyval(p,x);
plot(d,v,'o',x,y)
xlabel('diameter')
ylabel('volume')
legend('observed data','model prediction')
```

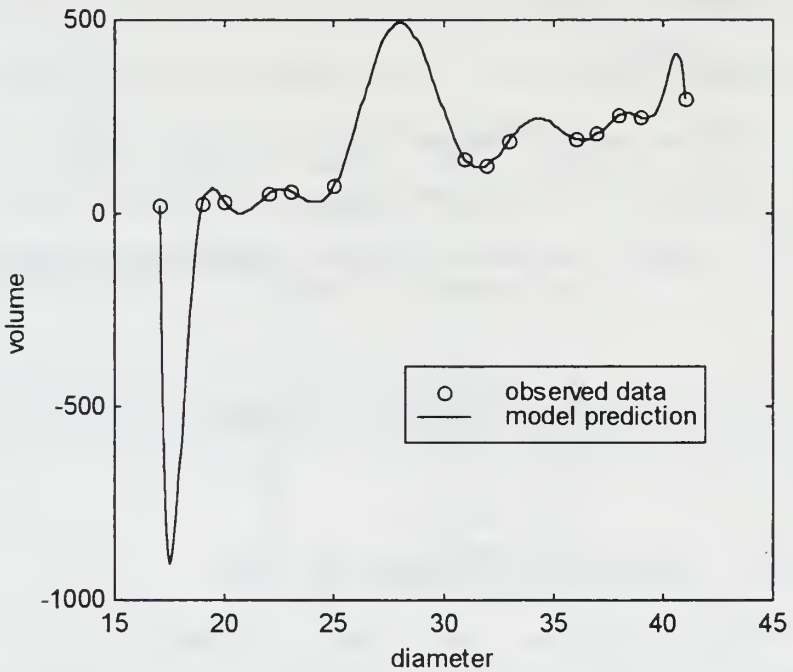

Figure IV.6. Ponderosa Pine tree data with 13th degree polynomial

Although the polynomial model does pass exactly through each of the data points, it does a poor job predicting the volume of trees having diameters between 17

and 19 inches and between 25 and 31 inches. This example demonstrates a disadvantage of using high-order polynomial models. These polynomials tend to oscillate, especially near the end points of the data, rendering them almost useless for interpolation between data points.

The next method we will consider eliminates most of the disadvantages associated with high-order polynomials.

2. Low-Order Polynomial Models

Unlike high-order polynomials, low-order polynomials generally do not pass exactly through every data point. Rather, in most cases they tend to "smooth" the data in providing a viable model for purposes of interpolation and extrapolation. Again, the MATLAB function **polyfit** can be used to obtain these low-order polynomials.

To demonstrate this process, consider again the problem of predicting elapsed time of a tape recorder.

a) Elapsed Time of a Tape Recorder Revisited

Recall the tape recorder data:

c	100	200	300	400	500	600	700	800
t	205	430	677	945	1233	1542	1872	2224

Before fitting a polynomial to this data, we need to know what order polynomial to use. A divided difference table, constructed from the data, can assist in the determination of the order of the polynomial to use (Section

6.3 of (2) explains the use of divided difference tables to determine the order of an interpolating polynomial). The following M-file produces a divided difference table for the tape recorder data. Table IV.1 depicts the divided difference table.

```
c=100:100:800;
t=[205 430 677 945 1233 1542 1872 2224];
D=divdiff(c,t,4);
dlmwrite('f:\thesis\divtab',D,'\t')
```

We introduce a new function in this file called **divdiff**. The command **divdiff(x,y,ord)** produces a divided difference table of order "ord" ("ord" being the highest divided difference calculated). This function is not available in MATLAB; it can be found in the appendix.

x_i	y_i	Δ	Δ^2	Δ^3	Δ^4
100	205	2.25	0.00110	0.00000	0.00000
200	430	2.47	0.00105	0.00000	0.00000
300	677	2.68	0.00100	0.00000	0.00000
400	945	2.88	0.00105	0.00000	0.00000
500	1233	3.09	0.00105	0.00000	
600	1542	3.30	0.00110		
700	1872	3.52			
800	2224				

Table IV.1. Divided difference table for tape recorder data

We see in Table IV.1 that the second divided differences are virtually constant, and the third divided differences are zero (to five decimal places). This suggests that the data is essentially quadratic (again, see

Section 6.3 of (2) to see why this is true), which leads us to attempt to fit a second-order polynomial to the data.

Fitting a second-order polynomial to the data can be accomplished with the following M-file:

```
c=100:100:800;
t=[205 430 677 945 1233 1542 1872 2224];
p=polyfit(c,t,2);
x=0:1000;
y=polyval(p,x);
plot(c,t,'o',x,y)
xlabel('counter reading')
ylabel('elapsed time (sec)')
legend('observed data','model prediction')
```

The plot in Figure IV.7 shows that the second-order polynomial model does a very good job of capturing the trend of the data even though the residual plot shown in Figure IV.8 reveals that the model does not pass exactly through each data point.

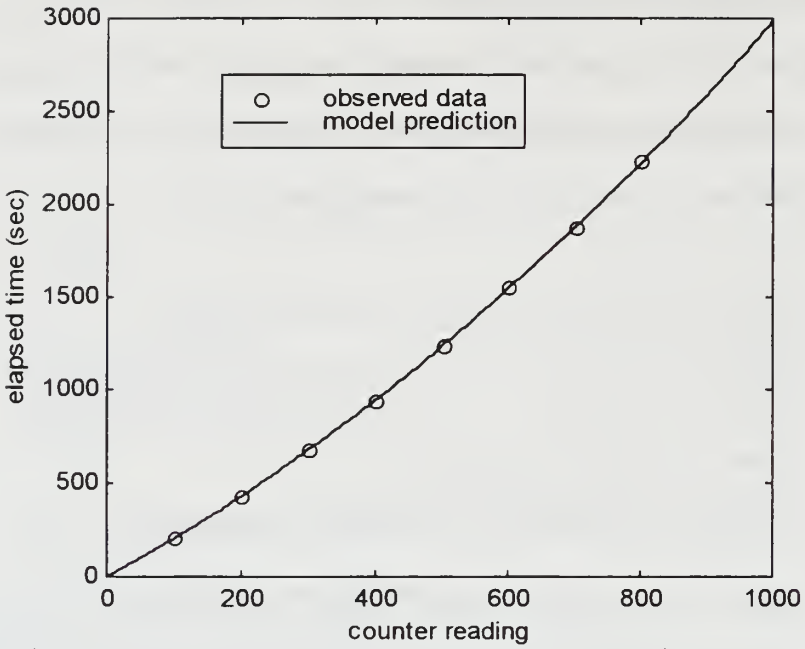

Figure IV.7. Second-order polynomial model

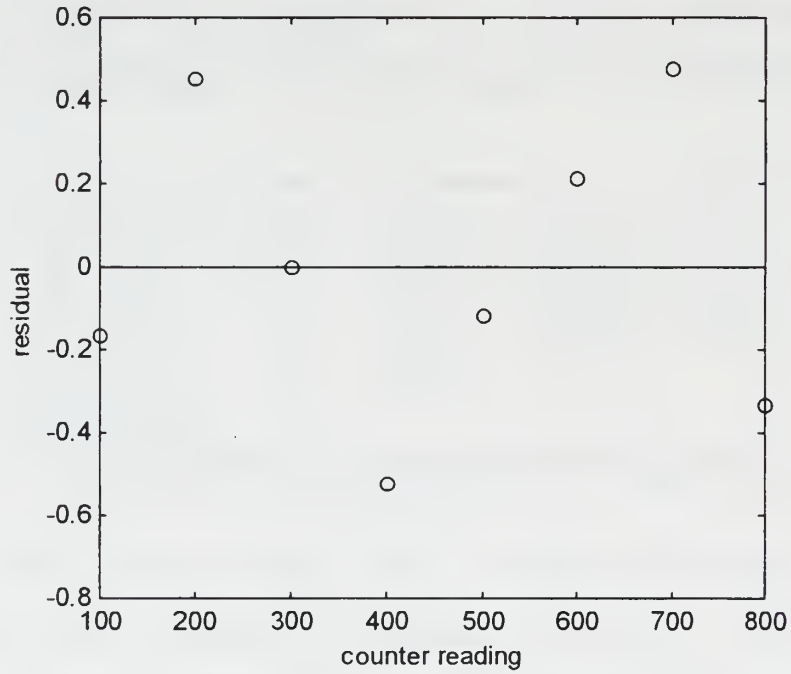

Figure IV.8. Residuals from 2nd-order polynomial model

Figure IV.8 reveals how good the quadratic fit really is. The largest error shown in this plot is less than one second; a comparison of the error with the magnitude of the dependent variable (elapsed time), which is measured in *thousands* of seconds, results in a largest relative error of about one-tenth of one percent. This certifies the accuracy of the quadratic model.

Let's look at another example that will further demonstrate the advantages of using low-order polynomials.

b) Volume of a Ponderosa Pine

Recall that previously we attempted to model the volume of Ponderosa Pine Trees with a 13^{th}-degree polynomial. Figure IV.6 demonstrates that, although this high-order polynomial does pass exactly through each data point, it might be better to model the situation with a low-order polynomial. To determine the order of the polynomial that would best fit the data, we first produce a divided difference table (using the Ponderosa Pine Tree data previously given) with the M-file shown below:

```
d=[17 19 20 22 23 25 31 32 33 36 37 38 39 41];
v=[19 25 32 51 57 71 141 123 187 192 205 252 248 294];
D=divdiff(d,v,4);
dlmwrite('f:\thesis\divtab',D,'\t')
```

This M-file produces Table IV.2.

d	V	Δ	Δ^2	Δ^3	Δ^4
17	19	3.0000	1.3333	-0.1000	-0.0667
19	25	7.0000	0.8333	-0.5000	0.1333
20	32	9.5000	-1.1667	0.3000	-0.0247
22	51	6.0000	0.3333	0.0278	-0.0563
23	57	7.0000	0.5833	-0.5357	0.6191
25	71	11.6667	-4.2338	5.6548	-1.5429
31	141	-18.0000	41.0000	-11.3167	2.5000
32	123	64.0000	-15.5833	3.6833	-0.1417
33	187	1.6667	2.8333	2.8333	-2.8333
36	192	13.0000	17.0000	-14.1667	4.5583
37	205	47.0000	-25.5000	8.6250	
38	252	-4.0000	9.0000		
39	248	23.0000			
41	294				

Table IV.2. Divided difference table for Ponderosa Pine Tree data

As it turns out, the divided differences in Table IV.2 provide very little useful information to help determine the degree of polynomial to use. Negative values (anywhere in the table) and large values in the high order divided differences (3^{rd} or 4^{th}) tend to result from errors and irregularities in the observed data. We resort to an educated guess and try a quadratic model. This M-file generates the quadratic model as well as the scatterplot at Figure IV.9 and the residual plot at Figure IV.10:

```
d=[17 19 20 22 23 25 31 32 33 36 37 38 39 41];
v=[19 25 32 51 57 71 141 123 187 192 205 252 248 294];
p=polyfit(d,v,2);
x=17:.1:41;
y=polyval(p,x);
plot(d,v,'o',x,y)
xlabel('diameter')
ylabel('volume')
legend('observed data','model prediction')
figure(2)
plot(d,polyval(p,d)-v,'o',d,zeros(size(d)))
xlabel('diameter')
ylabel('residual')
```

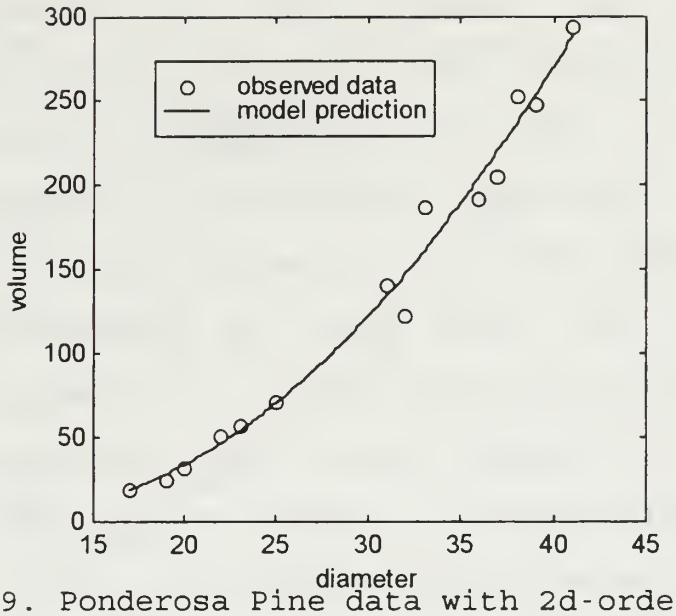

Figure IV.9. Ponderosa Pine data with 2d-order polynomial model

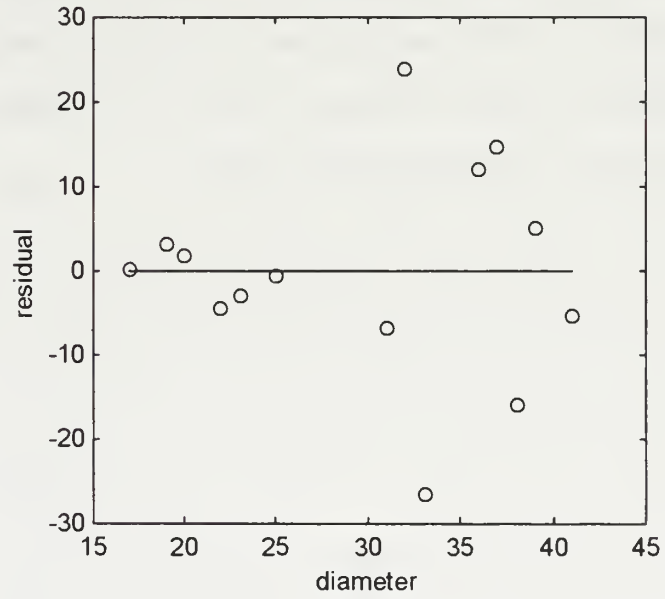

Figure IV.10. Residuals from 2d-order polynomial model

The model shown in Figure IV.9 seems to capture the trend of the data more accurately than the model shown in Figure IV.6. To illustrate, suppose we desire to know the volume of a 28-inch diameter tree. According to 13^{th}-degree polynomial model, the volume of this tree is expected to be almost 494 board feet, an unreasonable estimate (compare this to the actual data: a 31-inch diameter tree yielded only 141 board feet). On the other hand, the quadratic model predicts a volume of just under 100 board feet, a much more believable number.

The errors plotted in Figure IV.10 also seem to indicate that the quadratic is a good fit. For comparison, let's try modeling this data with a cubic polynomial. The M-file (not shown) that generates the cubic model (Figure IV.11) and residual plot (Figure IV.12) is nearly identical to the one used for the quadratic fit.

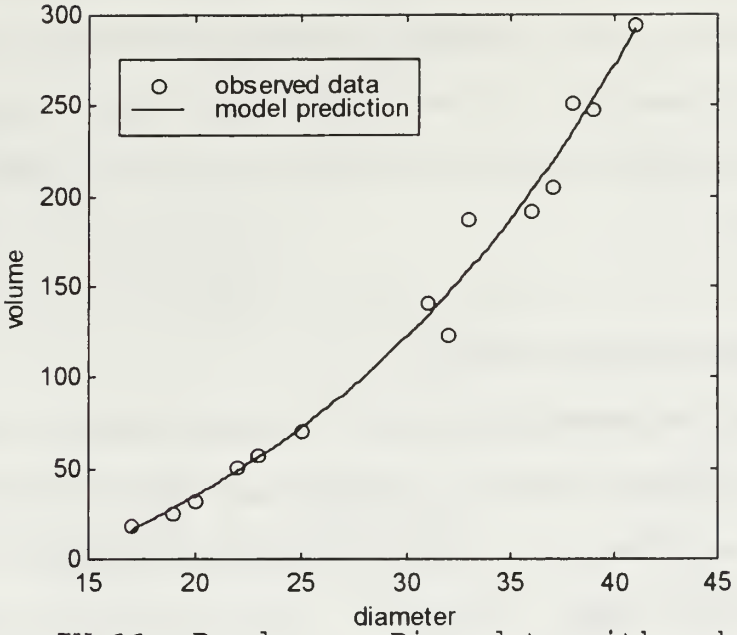

Figure IV.11. Ponderosa Pine data with cubic model

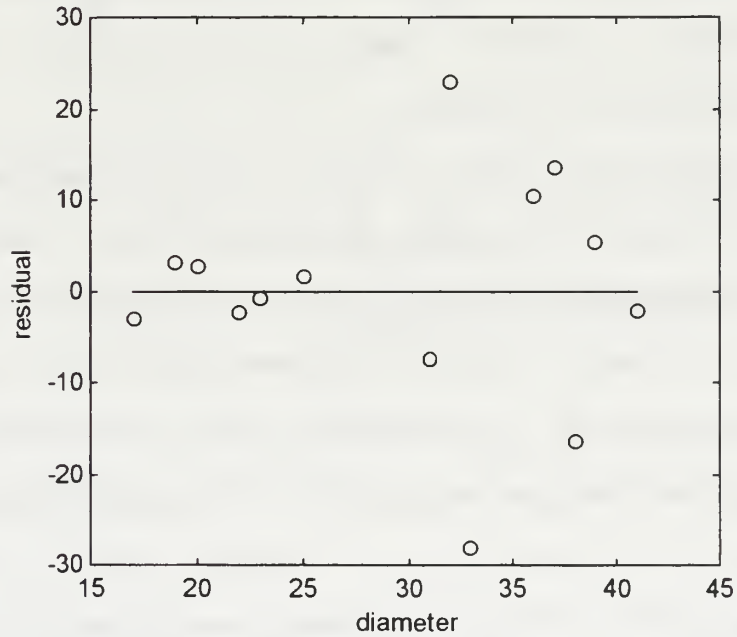

Figure IV.12. Residuals from cubic model

There does not seem to be any significant difference (in terms of accuracy) in the two models.

However, one might be inclined to select the cubic model over the quadratic for reasons of proportionality. Diameter is a linear measure, and we know (see Bass Fishing Derby example in Chapter III) that volume is proportional to the length *cubed*.

3. Cubic Spline Models

Cubic spline models combine the advantage of high-order polynomials to pass exactly through each data point with the feature of low-order polynomials (data smoothing) to capture the trend of the data. However, while high- and low-order polynomial models can be useful to both extrapolate and interpolate, cubic spline models are generally only useful for interpolation purposes (see Section 6.4 of (2) for a detailed discussion of splines).

MATLAB constructs cubic splines with its **spline(x,y,xx)** function. The syntax is **yy = spline(x,y,xx)**, where *x* and *y* are vectors representing the data to be modeled, *xx* is the new abscissa (x-axis) vector representing the range over which the spline will be evaluated, and *yy* is the output vector containing the value of the spline function at each point in *xx*.

Let's consider again the Vehicular Stopping Distance problem for illustrative purposes. The M-file below demonstrates how to use MATLAB to model total stopping

distance with a cubic spline (using mean total stopping distance data from Table III.1):

```
speed=20:5:80;
obs=[42 56 73.5 91.5 116 142.5 173 209.5 248 292.5 343 401 464];
x=20:.1:80;
y=spline(speed,obs,x);
plot(speed,obs,'ko',x,y)
xlabel('speed (mph)')
ylabel('total stopping distance (feet)')
legend('Observation','Cubic Spline')
```

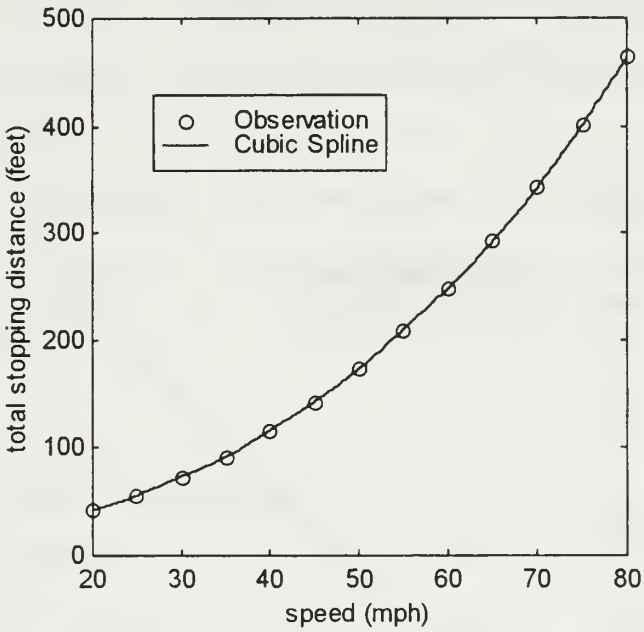

Figure IV.13. Total stopping distance modeled with cubic spline

Figure IV.13 shows the features of cubic spline models. Note the smoothness of the model and how it passes exactly through each data point. We could easily use this model to interpolate between data points. For example, a few modifications to the previous M-file, as noted below, can provide us the stopping distance predicted by the cubic

spline model for any speed within the domain of our data (i.e., between 20 and 80 mph). Figure IV.14 provides a visual demonstration of this capability.

```
speed=20:5:80;
obs=[42 56 73.5 91.5 116 142.5 173 209.5 248 292.5 343 401 464];
x=20:.1:80;
y=spline(speed,obs,x);
y1=spline(speed,obs,28);
plot(speed,obs,'ko',x,y,28,y1,'+')
xlabel('speed (mph)')
ylabel('total stopping distance (feet)')
legend('Observation','Cubic Spline')
text(28,50,['stopping distance at 28'])
text(28,20,['mph is ',num2str(y1),' ft.'])
```

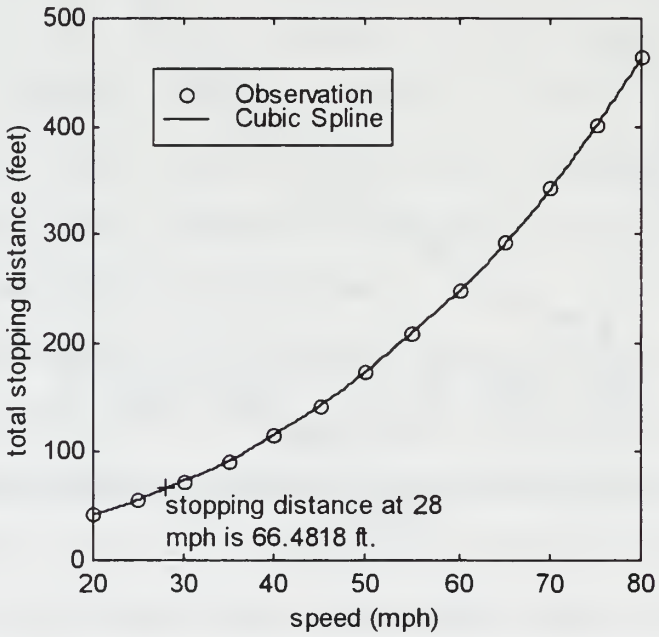

Figure IV.14. Cubic spline model used for interpolation

V. SIMULATION MODELING

In situations where either the modeler cannot construct an adequate analytic model to explain the behavior being observed or the behavior itself is probabilistic in nature, Monte Carlo simulation may provide a useful approach. In this chapter, we demonstrate MATLAB's capabilities for simulation modeling by means of several examples. We first cover random number generation, followed by simulations of both deterministic and probabilistic behaviors. In this process, we introduce the MATLAB function **random** and demonstrate the use of **logical operators** in MATLAB. Chapter 7 of (2) contains a more detailed discussion of simulation modeling.

A. RANDOM NUMBER GENERATION IN MATLAB

Random numbers can be generated in MATLAB in many different ways using several different functions. The function we use exclusively for this chapter is **random**. The syntax for using this function is

r = random('name',a,b,m,n)

where name represents the name of the distribution desired (i.e., 'unif' for uniform, 'norm' for normal, and 'bino' for binomial, to name just 3 of the 20 distributions available); a and b represent the parameters of the desired distribution; and m and n represent the dimensions of the

desired output (i.e., m = n = 1 will result in a scalar output of one random number; m = 2, n = 3 will result in a 2x3 matrix of random numbers; and so forth). We use the Uniform distribution exclusively in this presentation.

To demonstrate the use of this function, a simple example is in order. The input:

```
» r=random('norm',0,1,1,5)
```

results in the output:

```
r =
   -0.4326   -1.6656    0.1253    0.2877   -
1.1465
```

which consists of a five-element row vector containing random samples from a normal distribution with mean zero and standard deviation one. Obviously, since this is a random number generator, different results are achieved when the same command is entered again. With this command, MATLAB is capable of quickly generating enormous arrays of random numbers, making it a necessary and convenient tool for conducting simulations.

We now look at several examples.

B. SIMULATING DETERMINISTIC BEHAVIOR

1. Area Under a Curve

In this section, we demonstrate the use of Monte Carlo simulation to model a deterministic behavior: the area under a positive curve y = f(x). First identify any number M such that $0 \leq f(x) \leq M$ over the closed interval [a,b].

is in the interval [a,b] and y_i is in the interval [0,M], for i = 1,2,...,n. Then count the number of points (x_i, y_i) that fall *under* the curve, and call this number **count**. Finally, calculate the area under the curve to be (approximately)

$$AREA = M(b-a)count/n.$$

A formal algorithm for this process can be found on page 221 of (2).

The following M-file computes the area under the curve $y = cos(x)$ over the interval

$$-\pi/2 \le x \le \pi/2$$

using M = 1:

```
count=0;
a=-pi/2; b=pi/2; M=1;
n=100;
x=random('unif',a,b,n,1);
y=random('unif',0,M,n,1);
z=y<=cos(x);
count=sum(z);
area=(b-a)*M*count/n
```

This file creates two n-long vectors, x and y, of random numbers drawn from the uniform distribution. The x values are chosen from the interval

$$-\pi/2 \le x \le \pi/2,$$

and the y values are taken from [0,1]. Once these vectors are created, the MATLAB **logic operator** "<=" creates the vector z: the line "z=y<=cos(x);" can be translated, "Test each x and y pair; if y is less than or equal to $cos(x)$,

assign the value one to the corresponding element in *z*. Otherwise, assign the value zero to the corresponding element in *z*." The sum of the elements in the vector *z* then corresponds to the number of random points from our sample that fall on or below the curve. The area is then computed as indicated in the last line of the above M-file.

Note the power of using logic operators in MATLAB. An alternative is to use "for" and "if" loops, but they significantly increase the time required for the simulation (for large *n* (in the neighborhood of 10,000), the time required for running this simulation using loops instead of logic operators can be measured in *minutes*, whereas with logic operators, the time required is decreased by a factor of about 25). A general rule of thumb for any MATLAB code is to avoid using loops whenever possible.

Running the above M-file for various values of n produces the following results:

n	approx. area
1000	2.0106
2000	2.0122
3000	1.9855
4000	2.0169
5000	2.0062
6000	2.0070
7000	1.9940
8000	1.9977
9000	1.9726
10000	1.9984

Table V.1. Monte Carlo approximation to the area under the curve $y = cos(x)$

We can compare these results with the actual area, which is 2 square units and see that our simulation model is indeed reasonable. Note that increasing n does not necessarily result in greater accuracy.

Figure V.1 provides a visual representation of how the Monte Carlo simulation works (the simulation equates to throwing darts randomly at a dart board).

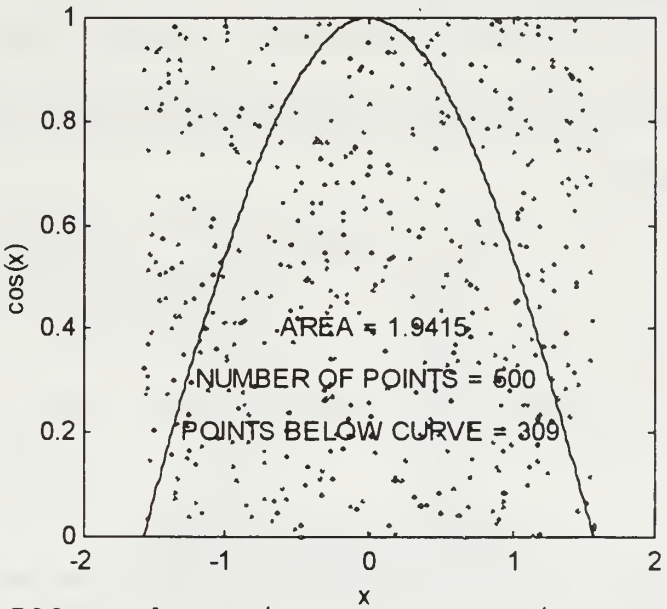

Figure V.1. 500 random points to approximate the area under the curve $y = cos(x)$

Let's look at another example of a deterministic behavior modeled with Monte Carlo simulation.

2. Volume Under a Surface

Suppose we wanted to determine the volume of the sphere

$$X^2 + y^2 + z^2 \leq 1$$

that lies in the first octant, x > 0, y > 0, z > 0. We can approach this problem in much the same way as for the area under a curve problem (see page 223 of (2) for a formal algorithm) by using the M-file below:

```
a=0; b=1; c=0; d=1; M=1;
n=100000;
x=random('unif',a,b,n,1);
y=random('unif',c,d,n,1);
z=random('unif',0,M,n,1);
v=(x.^2+y.^2+z.^2)<=1;
count=sum(v);
area=(b-a)*(d-c)*M*count/n
```

Running this code for various values of n produces the following results:

n	Approx. vol.
1000	0.52600
2000	0.54550
3000	0.52933
4000	0.52575
5000	0.51940
6000	0.51767
7000	0.52429
8000	0.53312
9000	0.52233
10000	0.52430

Table V.1. Monte Carlo approximation to the volume in the first octant under the unit sphere

The actual volume is 0.5236 to four decimal places. Again, notice that increasing the number n of random points does not necessarily result in increased accuracy.

C. SIMULATING PROBABILISTIC BEHAVIOR

1. Tossing a Fair Coin

It is generally understood that the probability of obtaining a head in tossing a fair coin is 0.5. This does

not mean, however, that one out of every two tosses results in a head. It only means that in the long run the ratio of heads obtained to the number of coins tossed should be close to 0.5.

To simulate *n* coin tosses for any value of *n*, we need only draw *n* uniformly distributed random numbers from the interval [0,1] and count the number of samples less than 0.5. This count value represents the number of heads obtained. For a formal algorithm, see page 229 of (2). The M-file below performs this simulation:

```
n=1000;
x=random('unif',0,1,n,1);
y=x<=0.5;
numheads=sum(y)
percheads=numheads/n
```

Running this code for various values of n produces the following results:

Number of Tosses	Number of Heads	Percent Heads
1000	504	0.50400
2000	1012	0.50600
3000	1523	0.50767
4000	1998	0.49950
5000	2476	0.49520
6000	2990	0.49833
7000	3492	0.49886
8000	3996	0.49950
9000	4545	0.50500
10000	5099	0.50990

Table V.3. Results of tossing a fair coin

This next example for simulating a roll of a fair die is more complex.

2. The Roll of a Fair Die

Instead of only two possible outcomes, as in tossing a coin, we now must simulate an event with six possible outcomes. In order to do this, we modify our code to distribute the count of random values between zero and one into six "bins." The formal algorithm for this simulation is on page 229 of (2). Our modified M-file for this simulation appears below.

```
n=1000;
x=random('unif',0,1,n,1);
y1=x<=(1/6);
y2=x>(1/6) & x<=(2/6);
y3=x>(2/6) & x<=(3/6);
y4=x>(3/6) & x<=(4/6);
y5=x>(4/6) & x<=(5/6);
y6=x>(5/6);
num_ones=sum(y1)
num_twos=sum(y2)
num_threes=sum(y3)
num_fours=sum(y4)
num_fives=sum(y5)
num_sixes=sum(y6)
```

Notice the use of *compound* logic statements. As an example, the line

```
y2=x>(1/6) & x<=(2/6);
```

can be translated, "Create a vector called y2 whose elements are equal to one for every x such that $1/6 < x \leq 2/6$ and zero otherwise." Simply summing every element of this vector (using the **sum** command) gives the number of "twos" rolled in this simulation.

Running this code for several values of *n* produced the following results:

Number of Rolls, *n*	Percent Ones	Percent Twos	Percent Threes	Percent Fours	Percent Fives	Percent Sixes
10	0.10000	0.20000	0.10000	0.20000	0.30000	0.10000
100	0.10000	0.22000	0.13000	0.19000	0.16000	0.20000
1000	0.17800	0.16100	0.20400	0.15300	0.15900	0.14500
10000	0.16880	0.17140	0.16040	0.16270	0.16520	0.17150
100000	0.16648	0.16708	0.16542	0.16520	0.16883	0.16699

Table V.4. Results from *n* rolls of a fair six-sided die

Comparing these results with the expected result of 1/6 for each entry shows that the model is indeed reasonable. For large *n*, the simulation results are close to the expected value.

VI. LINEAR PROGRAMMING

Linear programming (LP) is a branch of mathematics used to obtain an optimal (maximum or minimum) value for a predetermined linear objective that is subject to certain linear constraints. This technique is very useful for solving such problems as resource allocation, profit maximization, and transportation system optimization, to name just a few.

A typical LP consists of a set of **decision variables**, a linear **objective function**, and a set of linear **constraints**. We define these terms below:

- **Decision variables**: the parameters over which the decision-maker has control. For example, if a carpenter must decide how many chairs and tables to make to maximize his profit, his decision variables would be number of chairs to make (call this **X**) and number of tables to make (call this **Y**).

- **Objective function**: a linear combination of the decision variables to be maximized or minimized. For example, suppose our carpenter makes a net profit of $10 for chairs and $25 for tables, and he desires to maximize his net profit. His objective function would then be

$$\textit{Maximize } 10X + 25Y.$$

- **Constraints**: side conditions that must be met. The objective function must be maximized (or minimized) *subject to* these conditions. Continuing with our carpenter example, a possible constraint could be in the area of available materials. For example, suppose the carpenter has 250 board feet of wood available to make chairs and tables. If chairs require 17 board feet of wood and tables require 29, then the constraint for wood would be

$$17X + 29Y \leq 250.$$

Linear programs can be solved in many different ways using a variety of graphical, algebraic, and computational techniques. We will discuss two methods of solving such problems: geometric and the tableau Simplex methods.

A. GEOMETRIC SOLUTIONS

To demonstrate a geometric approach to solving linear programs, let's look at an example involving a different carpenter (taken from section 9.1 of (2)).

EXAMPLE: The Carpenter Problem.

Scenario: A carpenter makes tables and bookcases and sells them for a net profit of $25 and $30 each, respectively. He would like to determine how many of each to make each week in order to maximize his profit. He has 690 board feet of lumber available each week and up to 120 hours of labor. He estimates that tables require 20 board feet of lumber and 5 hours of labor to complete, while bookcases require 30 board feet of lumber and 4 hours of labor.

The first step to solving a problem like this is to put all of this information into a workable format. So, we let X represent the number of tables to be produced and Y denote the number of bookcases. We can formulate the carpenter's problem as follows:

Maximize $25X + 30Y$ (objective function)

Subject to:

$20X + 30Y \leq 690$ (lumber constraint)

$5X + 4Y \leq 120$ (labor constraint)

$X, Y \geq 0$

Now, let's look at the geometric representation of this problem.

First, we plot the lumber constraint:

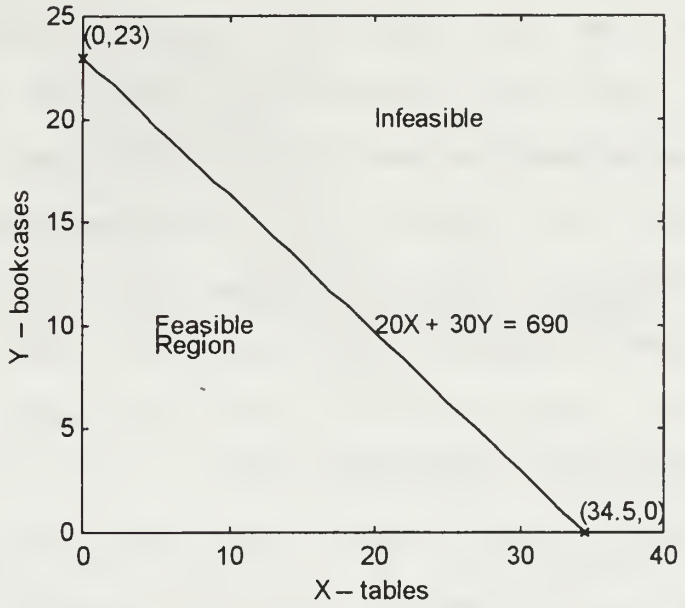

Figure VI.1. Geometric interpretation of the lumber constraint

The line *20X + 30Y = 690* represents the region in which the lumber constraint is *met at equality*. The phrase "*met at equality*" is used to describe a situation where a constraint is satisfied *exactly*. In this problem, for example, if the carpenter decided to make no tables and 23 bookcases, he would use *exactly* 690 board feet of lumber. So is the case of any other combination of tables and bookcases that lies along this line.

Note, however, that the constraint for lumber is an *inequality* constraint. This translates geometrically to (X,Y) combinations (combinations of tables and bookcases) that lie not only *on* the line, but also *anywhere to the left* of the line. We call this the **feasible region** for this constraint. Any (X,Y) combination lying in the feasible region is guaranteed to meet the lumber constraint. To check, select the point (10,5) (corresponding to the potential decision to make 10 tables and 5 bookcases). It is easy to see that this point lies to the left of the line in Figure VI.1. We can also see that the lumber required by this table/bookcase combination is

$$20(10) + 30(5) = 200 + 150 = 350 \text{ board feet}$$

which is less than the maximum available lumber of 690 board feet per week. We see that the point (10,5) is indeed feasible as far as the lumber constraint is concerned.

Note that lumber is not the only constraint; we also have a constraint for available labor. We now plot the labor constraint along with that for lumber.

The feasible region for both constraints together, shown in Figure VI.2, is a bit smaller than the one in Figure VI.1. In order for a point to be feasible, it must satisfy both constraints. Only those (X,Y) combinations that lie inside this feasible region do so. This region turns out to be a **convex set** (also known as a **polygon**).

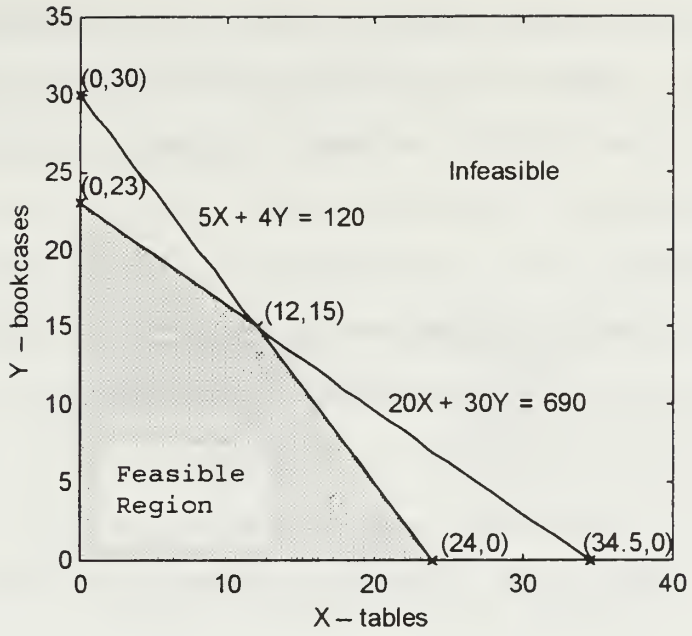

Figure VI.2. Lumber and labor constraints

The final two constraints in our problem indicate that the decision variables X and Y must be nonnegative. That is, (X,Y) combinations must lie above the X-axis and to the right of the Y-axis in order to be feasible. The feasible region (shaded) shown in Figure VI.2 already accounts for these two constraints.

Now that we have plotted the feasible region, we have narrowed our search for a solution somewhat. We know that the solution must lie inside (or on the border of) the feasible region, since this is the only geometric location where all the constraints are met. There are an infinite number of points within the feasible region; fortunately, however, it turns out that an optimal solution to a linear

program, if one exists, lies on one of the **extreme points** of the convex set formed by the intersection of the set of constraints. The values of the objective function at the extreme points are

Extreme point	Objective function value
(0,0)	$0
(24,0)	$600
(12,15)	$750
(0,23)	$690

Table VI.1. Objective function evaluated at the extreme points of the feasible region

So, to maximize his profit, the carpenter should make 12 tables and 15 bookcases. This combination earns him $750 per week, and there exists no other combination that would earn him more. This is the **optimal solution** to this linear program.

B. TABLEAU SIMPLEX METHOD

As the name suggests, the tableau Simplex Method of solving linear programs involves arranging the coefficients of the objective function and constraints in a table, or **tableau**. For a complete description of this method, see Section 9.3 of (2). To demonstrate this method, we will again use the Carpenter's Problem.

The appendix contains an M-file called tableau.m that serves as an interactive tableau-based LP solver. We will

demonstrate the use of this program as we go along. Let us begin with the necessary format for the LP.

In order to use the tableau method to solve a linear program, the LP must be expressed in standard **tableau format**. Tableau format assumes that the objective function is to be maximized and the constraints are "less-than-or-equal-to" inequalities. Additionally, all variables are nonnegative. The Carpenter's Problem, which is to

$$\text{Maximize } 25X + 30Y$$

Subject to

$$20X + 30Y \leq 690$$

$$5X + 4Y \leq 120$$

$$X, Y \geq 0$$

is already in the desired format.

Prior to solving with the tableau, one final modification to the LP is needed. We first constrain the objective function to be no worse than its current value (assumed to be zero to start) and express this idea with the less-than-or-equal-to constraint

$$-25X - 30Y \leq 0.$$

We then add nonnegative **slack variables** to all constraints so as to transform them into **equality** constraints. This process results in the **augmented constraint set**

$$20X + 30Y + a = 690$$

$$5X + 4Y + b = 120$$

$$-25X - 30Y + z = 0$$

where all variables are nonnegative. The value of the variable z represents the value of the objective function.

Now that we have the correct format, let us begin using *tableau.m* to solve this problem.

The M-file *tableau.m* is executed by typing the word "tableau" at the prompt in the MATLAB Command Window. The first screen that appears is a description of the program and instructions for its use. Figure VI.3 shows what appears in the MATLAB Command Window when *tableau* is executed.

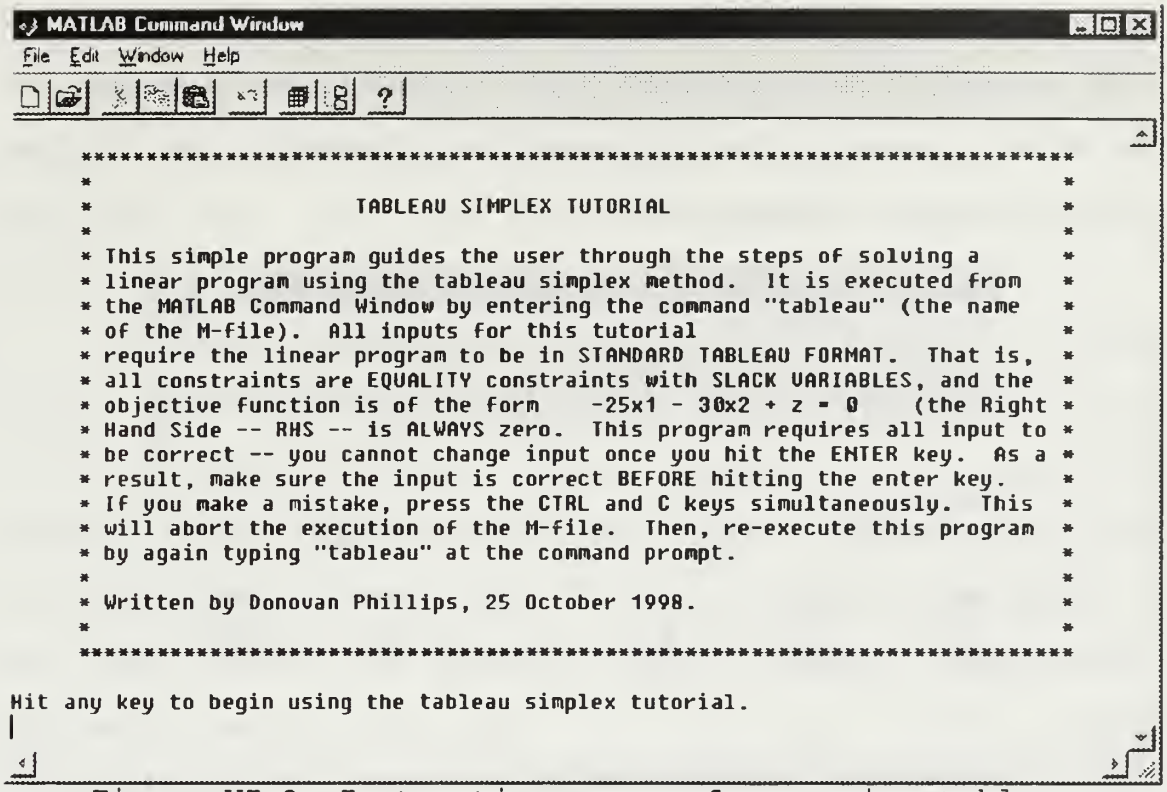
MATLAB Command Window
File Edit Window Help
TABLEAU SIMPLEX TUTORIAL
This simple program guides the user through the steps of solving a linear program using the tableau simplex method. It is executed from the MATLAB Command Window by entering the command "tableau" (the name of the M-file). All inputs for this tutorial require the linear program to be in STANDARD TABLEAU FORMAT. That is, all constraints are EQUALITY constraints with SLACK VARIABLES, and the objective function is of the form -25x1 - 30x2 + z = 0 (the Right Hand Side -- RHS -- is ALWAYS zero. This program requires all input to be correct -- you cannot change input once you hit the ENTER key. As a result, make sure the input is correct BEFORE hitting the enter key. If you make a mistake, press the CTRL and C keys simultaneously. This will abort the execution of the M-file. Then, re-execute this program by again typing "tableau" at the command prompt.
Written by Donovan Phillips, 25 October 1998.
Hit any key to begin using the tableau simplex tutorial.

Figure VI.3. Instruction screen for running tableau

On the next and subsequent screens, we are prompted to enter the necessary information with which the tableau will be constructed. Figure VI.4 shows the first such screen.

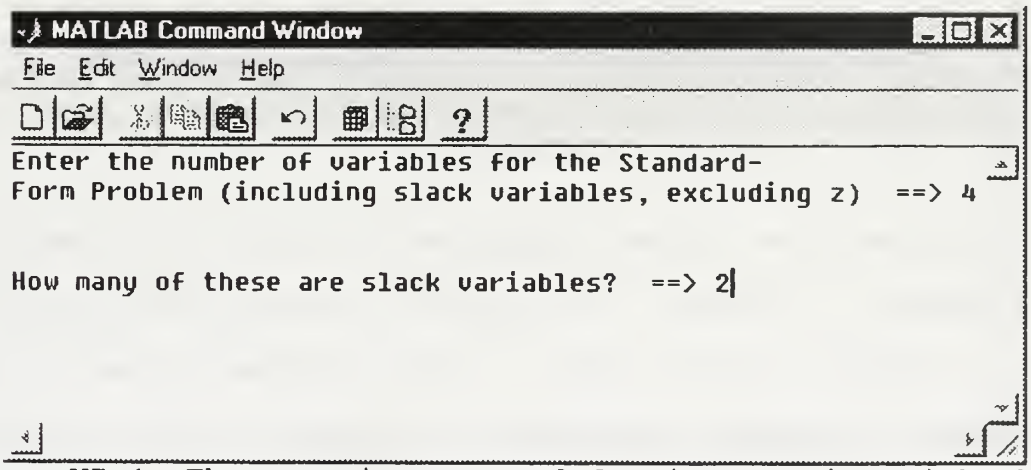
MATLAB Command Window
File Edit Window Help
Enter the number of variables for the Standard-
Form Problem (including slack variables, excluding z) ==> 4
How many of these are slack variables? ==> 2

Figure VI.4. The user is prompted for information with which to build the tableau

This process continues until all necessary information has been entered. The program then displays the initial tableau, shown in Figure VI.5.

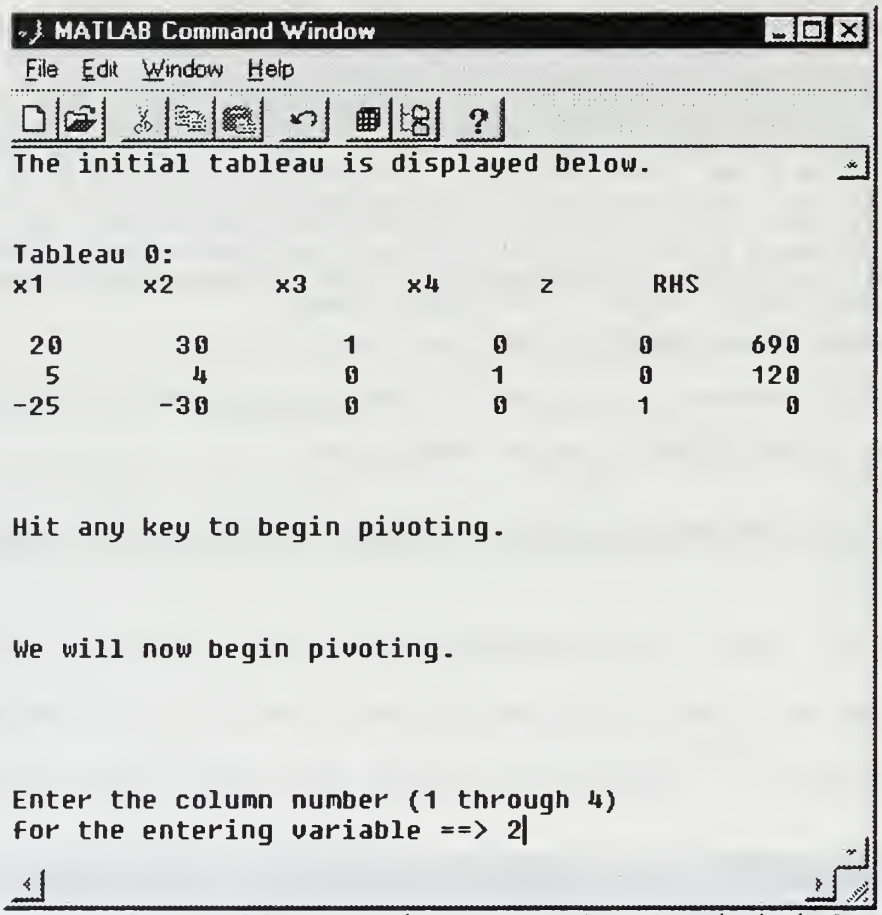
MATLAB Command Window
File Edit Window Help
The initial tableau is displayed below.
Tableau 0:
x1 x2 x3 x4 z RHS
20 30 1 0 0 690
5 4 0 1 0 120
-25 -30 0 0 1 0
Hit any key to begin pivoting.
We will now begin pivoting.
Enter the column number (1 through 4)
for the entering variable ==> 2

Figure VI.5. After all data is entered, the initial tableau is displayed and the user is prompted to choose an entering variable

Note that the tableau displayed in Figure VI.5 portrays the Carpenter's Problem. The initial extreme point implied by this tableau is the origin. The variables x1 and x2 are independent variables assigned the value 0; the variables x3, x4, and z are dependent variables whose values are to be

determined (x3 and x4 are the original slack variables corresponding to the two constraints in the problem). Each row in this tableau corresponds to a constraint from the problem, with the last row representing the objective function.

To pivot, we must select an entering variable. At this point, either x1 or x2 could enter, since their coefficients in the objective function are both negative (indicating that either variable could improve the current objective function value). We will choose x2 as the entering variable since it has the largest (in absolute value) negative coefficient. *Tableau* prompts the user to enter the column number corresponding to the variable chosen as the entering variable (column 2 in this case). An intermediate tableau is then displayed, as shown in Figure VI.6.

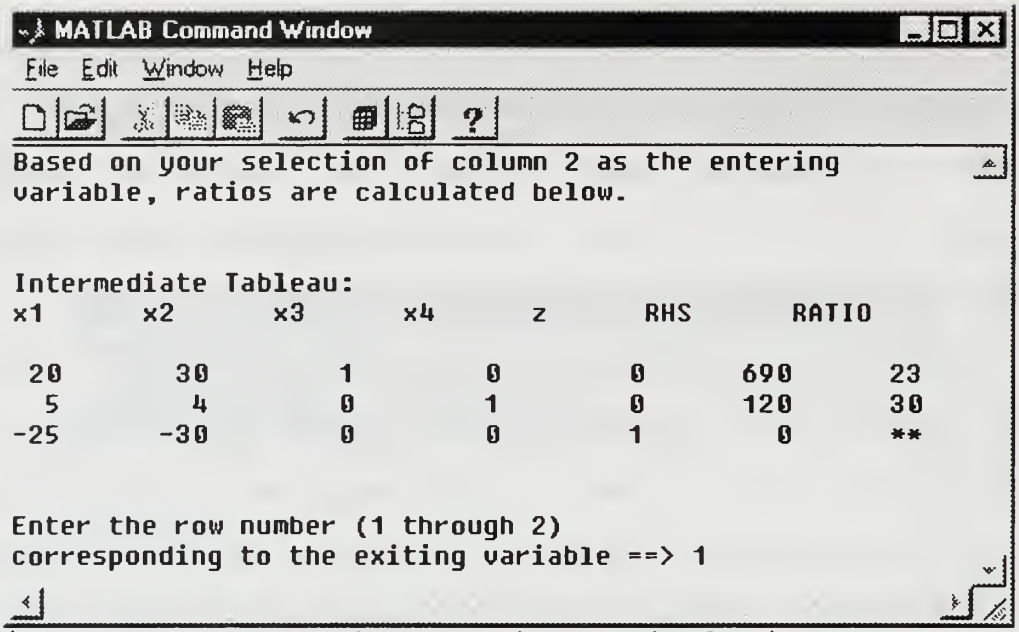
MATLAB Command Window
File Edit Window Help
Based on your selection of column 2 as the entering variable, ratios are calculated below.
Intermediate Tableau:
x1 x2 x3 x4 z RHS RATIO
20 30 1 0 0 690 23
5 4 0 1 0 120 30
-25 -30 0 0 1 0 **
Enter the row number (1 through 2) corresponding to the exiting variable ==> 1

Figure VI.6. Once the entering variable is chosen, an intermediate tableau is displayed and the user is prompted to choose an exiting variable

Tableau automatically computes the ratios necessary for conducting the feasibility test to choose the exiting variable. We choose x3 as the exiting variable since its corresponding ratio is smallest in value. We indicate this choice by entering the number **1** (corresponding to row 1) at the prompt in Figure VI.6. Tableau then pivots by performing the necessary row operations and displays the updated tableau, shown in Figure VI.7.

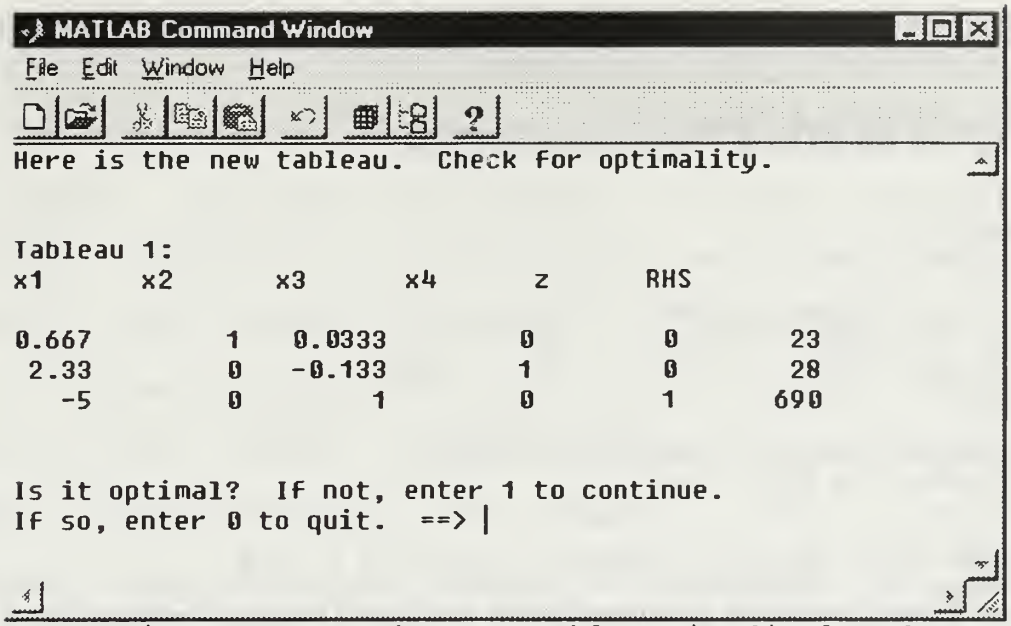
MATLAB Command Window
File Edit Window Help
Here is the new tableau. Check for optimality.
Tableau 1:
x1 x2 x3 x4 z RHS
0.667 1 0.0333 0 0 23
2.33 0 -0.133 1 0 28
-5 0 1 0 1 690
Is it optimal? If not, enter 1 to continue.
If so, enter 0 to quit. ==>

Figure VI.7. The new tableau is displayed

The value of the objective function at this point is 690, but we can see that this is not optimal because there is still a negative coefficient (of -5) in the objective function line of the tableau.

Since optimality has not yet been achieved, we enter the number **1** (to continue pivoting) at the prompt in Figure VI.7. We will then be prompted to choose new entering and exiting variables resulting in a new tableau. The process continues until we determine that optimality has been achieved. At this point, the optimal objective value is displayed (see Figure VI.8), and we are offered the opportunity to run the program again if desired.

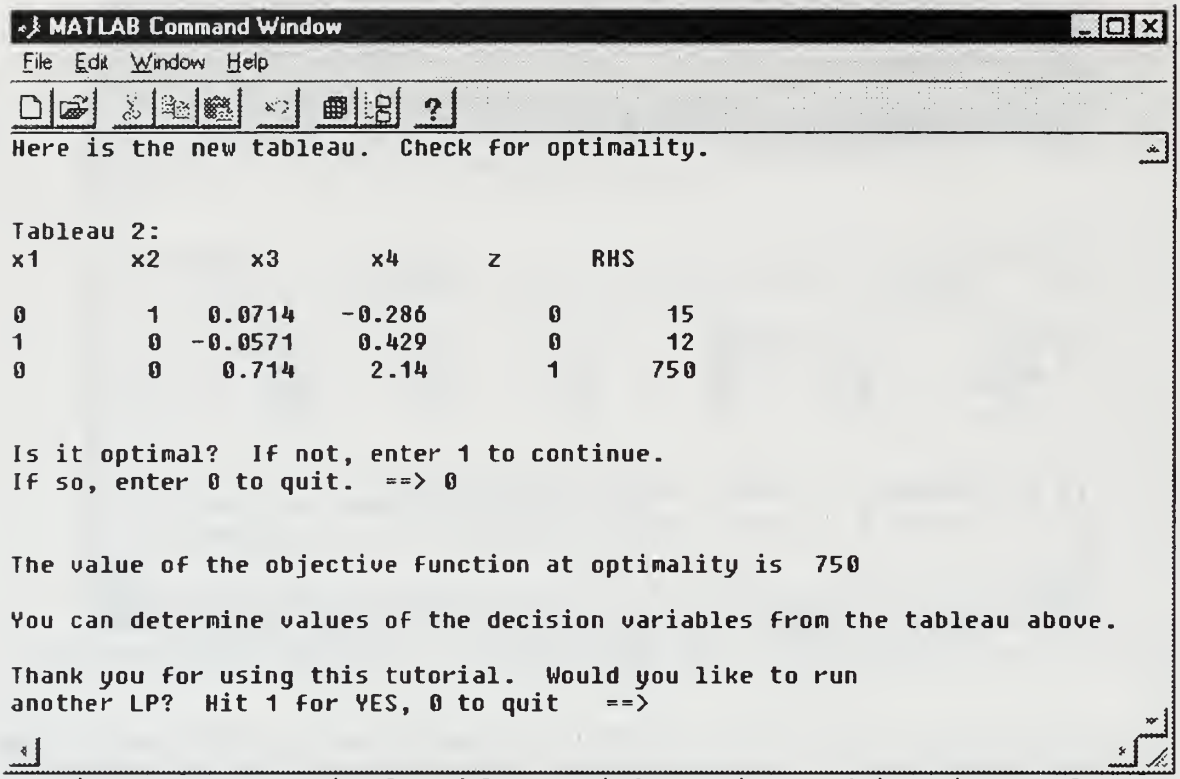
MATLAB Command Window
File Edit Window Help
Here is the new tableau. Check for optimality.
Tableau 2:
x1 x2 x3 x4 z RHS
0 1 0.0714 -0.286 0 15
1 0 -0.0571 0.429 0 12
0 0 0.714 2.14 1 750
Is it optimal? If not, enter 1 to continue.
If so, enter 0 to quit. ==> 0
The value of the objective function at optimality is 750
You can determine values of the decision variables from the tableau above.
Thank you for using this tutorial. Would you like to run
another LP? Hit 1 for YES, 0 to quit ==>

Figure VI.8. Final tableau with optimal objective value displayed

The M-file *tableau.m* can be found in the appendix. An Internet address is provided to make this file available for download and unrestricted use.

VII. CONCLUSION

In this thesis, we have demonstrated MATLAB's ability to handle the data requirements of many mathematical modeling scenarios. The inherent data manipulation, graphical and statistical capabilities of MATLAB make it an ideal software package for this type of work. MATLAB's easy-to-learn coding language enables the user to build models ranging from the most simple to some of the more complex, and, once the model is complete, to conduct critical sensitivity analysis with only minor modifications to the original code. This and other features make MATLAB the consummate platform with which to model and/or simulate most observable behaviors.

APPENDIX. FUNCTION M-FILES

M-files sited in the body of this thesis but not listed are included here. The first two, **rto(x,y)** and **cfit(x,y,n)**, use equation (5.7) on page 154 of (2) to fit one-term polynomial models. The third, **divdiff(x,y,order)** produces divided difference tables. The last, **tableau.m**, executes a tutorial on the Tableau Simplex Method.

All of these M-files are available for download at

http://math.nps.navy.mil/Archive

```
function b=rto(x,y)
      % rto(x,y) performs linear regression (of y on x)on a set of data
      % points, but forces the regression line to go through the origin
      % (thus the name rto -- "regression through the origin")
      %
      % rto(x,y) takes as input two equally-sized vectors, x and y,
      % and outputs the coefficient, b (as in y=bx), which creates the
      % best fit (in the least squares sense) line through the
      % origin.
      %
      % Written by Donovan Phillips, August 24, 1998
      %
      % initialization

      if nargin ~= 2
         error('Two vectors required as input.')
      end
      If length(x) ~= length(y)
         error('Vectors must be of equal length.')
      end

      % calculate coefficient b

      a=sum(x.*y);
      c=sum(x.^2);
      b=a/c;
```

```
function a=cfit(x,y,n)
% a = cfit(x,y) generates the coefficient "a" for the
% quadratic model
%                       y=ax^n
%
% where x and y are equal-length vectors representing
% the data being analyzed and n is the degree of the
% desired one-term polynomial.
%
% Written by Donovan Phillips, October 15, 1998.

% initialize

if nargin ~= 3
   error('Three input arguments required.')
end
if length(x) ~= length(y)
   error('Vectors must be of equal length.')
end

% compute a

b=x.^n;
c=b.*y;
d=x.^(2*n);
a=sum(c)/sum(d);
```

```
function D=divdiff(x,y,order)
% D = divdiff(x,y,order) generates a divided difference table with
% the highest order equal to "order" (the number of columns produced
% will be equal to order) for the data vectors x and y.  If no order
% is specified, the function will default to an order of 3.  This
% function calls the MATLAB function diff.m, which produces the
% standard difference of a vector.
%
% Written by Donovan Phillips, November 5, 1998.

% initialize
format bank

if ~isequal(size(x),size(y))
   error('Input vectors must be the same size and orientation (row or
column).')
end

if nargin==2
   order=3;
end

if size(x,1)==1
   x=x';
   y=y';
end

D=[];

% construct the divided difference table

D(:,1:2)=[x y]; % first 2 columns of the table

for n=1:order
   fill=zeros(n,1);
   denom=[];
   for i=1:(size(x,1)-n)
      denom(i)=x(i+n)-x(i);
   end
   a=D(1:(size(x,1)-n+1),n+1);
   y1=diff(a)./denom';
   D(:,n+2)=[y1;fill];
end
```

```
clc
v=['                                                                              '
   '     ********************************************************************************'
   '     *                                                                              *'
   '     *                      TABLEAU SIMPLEX TUTORIAL                                *'
   '     *                                                                              *'
   '     * This simple program guides the user through the steps of solving a           *'
   '     * linear program using the tableau simplex method.  It is executed from       *'
   '     * the MATLAB Command Window by entering the command "tableau" (the name       *'
   '     * of the M-file).  All inputs for this tutorial                                *'
   '     * require the linear program to be in STANDARD TABLEAU FORMAT.  That is,      *'
   '     * all constraints are EQUALITY constraints with SLACK VARIABLES, and the     *'
   '     * objective function is of the form    -25x1 - 30x2 + z = 0    (the Right    *'
   '     * Hand Side -- RHS -- is ALWAYS zero.  This program requires all input to     *'
   '     * be correct -- you cannot change input once you hit the ENTER key.  As a     *'
   '     * result, make sure the input is correct BEFORE hitting the enter key.        *'
   '     * If you make a mistake, press the CTRL and C keys simultaneously.  This      *'
   '     * will abort the execution of the M-file.  Then, re-execute this program     *'
   '     * by again typing "tableau" at the command prompt.                           *'
   '     *                                                                            *'
   '     * Written by Donovan Phillips, 25 October 1998.                              *'
   '     *                                                                            *'
   '     ********************************************************************************'];

disp(v)
disp(blanks(4))
disp('Hit any key to begin using the tableau simplex tutorial.')
pause

% determine size of tableau:

format short , clc
num_dgts=3;
disp('Enter the number of variables for the Standard-')
n=input('Form Problem (including slack variables, excluding z)  ==> ');
if n>8
   error('Too many variables (8 is max)')
end
fprintf('\n'), fprintf('\n')
s=input('How many of these are slack variables?  ==> ');

clc
disp('Enter the number of constraints')
m=input('(excluding nonnegativity constraints)   ==> ');
if m>4
   error('Too many constraints (4 is max)')
end

% construct the initial tableau:

tab=zeros(m+1,n+2);
numc=num2str(m); numv=num2str(n);

for i=1:m
   clc
   disp(['Enter the ',numv,' coefficients for constraint ',num2str(i)])
   z=input('(separated by spaces)  ==> ','s');
   tab(i,1:n)=str2num(z);

   fprintf('\n'), fprintf('\n')
   disp('Enter the value of the right hand side')
   tab(i,n+2)=input('(RHS) for this constraint  ==> ');
end

clc
disp(['Enter the ',num2str(n-s),' variable coefficients for the TABLEAU-FORMATTED'])
z=input('objective function (i.e., at least SOME should be NEGATIVE) ==> ','s');
tab(m+1,1:(n-s))=str2num(z);
```

```
tab(m+1,(n-s+1):n)=zeros(1,s);

tab(1:m,n+1)=zeros(m,1);
tab(m+1,n+1)=1;
tab(m+1,n+2)=0;

% initial tableau is now complete.

% display initial tableau:

clc
disp('The initial tableau is displayed below.')
fprintf('\n'), fprintf('\n')
disp('Tableau 0:')

varhold=['x1' 'x2' 'x3' 'x4' 'x5' 'x6' 'x7' 'x8'];

A=num2str(tab,num_dgts);
w=size(A,2)/size(tab,2); w=w-1.5; w=round(w/2);
varstr=[];
for j=1:n
   varstr(1,((j-1)*(2*w+2)+1):(j*(2*w+2)))=[varhold((2*j-1):2*j),blanks(2*w)];
end
varstr=setstr(varstr);
varstr=[varstr,'z',blanks(2*w),'RHS'];

disp(varstr)
fprintf('\n')
disp(A)
fprintf('\n'), fprintf('\n'), fprintf('\n')

% conduct pivoting

disp('Hit any key to begin pivoting.')
pause
fprintf('\n'), fprintf('\n'), fprintf('\n')
disp('We will now begin pivoting.')
fprintf('\n'), fprintf('\n')

status=1; count=0;
while status ~= 0
   count=count+1;
   fprintf('\n'), fprintf('\n')
   disp(['Enter the column number (1 through ',numv,')'])
   enter=input('for the entering variable ==> ');

   % compute RATIO and display intermediate tableau

   ratio=tab(:,n+2)./tab(:,enter);
   ratio=num2str(ratio,num_dgts);
   ratio(m+1,:)='*';
   clc
   disp(['Based on your selection of column ',num2str(enter),' as the entering'])
   disp('variable, ratios are calculated below.')
   fprintf('\n'), fprintf('\n')
   disp('Intermediate Tableau:')
   disp([varstr,blanks(2*w),'RATIO'])
   fprintf('\n')

   spc=[];
   for i=1:m+1
      spc(i,:)=blanks(2*w);
   end
   disp([num2str(tab,num_dgts),setstr(spc),ratio])
   fprintf('\n'), fprintf('\n')

   disp(['Enter the row number (1 through ',numc,')'])
   exit=input('corresponding to the exiting variable ==> ');
```

```
   fprintf('\n')

   % compute new tableau

   piv=tab(exit,enter);
   tab(exit,:)=tab(exit,:)/piv;
   for k=1:m+1
          if k ~= exit
             tab(k,:)=-tab(exit,:)*tab(k,enter) + tab(k,:);
          end
   end

   % display new tableau

   clc
   disp('Here is the new tableau.  Check for optimality.')
   fprintf('\n'), fprintf('\n')
   disp(['Tableau ',num2str(count),':'])

   A=num2str(tab,num_dgts);
   w=size(A,2)/size(tab,2); w=w-1.5; w=round(w/2);
   varstr=[];
   for j=1:n
      varstr(1,((j-1)*(2*w+2)+1):(j*(2*w+2)))=[varhold((2*j-1):2*j),blanks(2*w)];
   end
   varstr=setstr(varstr);
   varstr=[varstr,'z',blanks(2*w),'RHS'];

   disp(varstr)
   fprintf('\n')
   disp(A)

   % optimality decision

   fprintf('\n'), fprintf('\n')
   disp('Is it optimal?  If not, enter 1 to continue.')
   status=input('If so, enter 0 to quit.  ==> ');
end

% display results

fprintf('\n'), fprintf('\n')
disp(['The value of the objective function at optimality is  ',num2str(tab(m+1,n+2))])
disp(blanks(2))
disp('You can determine values of the decision variables from the tableau above.')
disp(blanks(3))

% option to run the tutorial again

disp(['Thank you for using this tutorial.  Would you like to run'])
q=input(['another LP?  Hit 1 for YES, 0 to quit   ==> ']);
if q == 1
   tableau
end
```

INDEX OF MATLAB TOPICS AND COMMANDS

MATLAB commands appear in bold type.

C

Curve fitting
- **cfit** 65, 66, 113, 114
- Divided differences
  - **divdiff** 76, 79, 113, 115
- **polyfit** 64, 68, 69, 70, 71, 72, 74, 75, 77, 80
- **polyval** 72, 74, 77, 80
- **rto** 29, 33, 51, 54, 57, 60, 65, 66, 113
- **spline** 64, 84, 85, 86

D

Data manipulation
- **max** 29
- **sum** 89, 92, 93, 94

F

File management
- **dlmwrite** 43, 76
- Set path 8

Functions 9

H

Help commands
- **help** 13
- **lookfor** 14

I

Interpolation ..... *See* Curve fitting

L

Linear programming
- **tableau.m** 104, 116

Logic operators 89
- compound 94

Looping
- **end** 18
- **for** 18, 20, 32, 115
- **if** 115, 118
- Looping index 22
- **while** 117

O

Operations 9

P

Plotting
Adding text
num2str 29, 54
text 29, 51
Labels
title 24
xlabel 24
ylabel 24
legend 25, 55, 65
Line types and colors 26
plot 22, 23, 24, 25
Punctuation and special characters
apostrophe 7
brackets 7
semicolon 7

R

Random number generation
random 87, 89, 93

V

Variable management
clear 12
whos 10

LIST OF REFERENCES

1. Hanselman, Duane C. and Littlefield, Bruce C., *Mastering Matlab 5: A Comprehensive Tutorial and Reference*, Prentice Hall, Upper Saddle River, New Jersey, 1997.

2. Giordano, Frank R., Weir, Maurice D., and Fox, William P., *A First Course in Mathematical Modeling*, 2nd Edition, Brooks/Cole Publishing Company, Pacific Grove, California, 1997.

3. Fox, William P., Girodano, Frank R., Maddox, Stephen L., and Weir, Maurice D., *Mathematical Modeling with Minitab*, Brooks/Cole Publishing Company, Belmont, California, 1987.

4. Beauchamp, Robert, *Mathematical Modeling Using Maple*, Master's Thesis, Naval Postgraduate School, Monterey, California, 1996.

5. Emmons, Nelson L., *Mathematical Modeling Using Microsoft Excel*, Master's Thesis, Naval Postgraduate School, Monterey, California, 1997.

6. Bertsimas, Dimitris and Tsitsiklis, John N., *Introduction to Linear Optimization*, Athena Scientific, Belmont, Massachusetts, 1997.

7. Hamilton, Lawrence C., *Regression with Graphics*, Duxbury Press, Belmont, California, 1992.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center. 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library. 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 94943-5101

3. Maurice D. Weir, Code MA/Wc. 3
   Naval Postgraduate School
   Monterey, California 93943-5216

4. Bard Mansager, Code MA/Ma. 1
   Naval Postgraduate School
   Monterey, California 93943-5216

5. Frank R. Giordano. 1
   P.O. Box 446
   Seaside, California 93955

6. Donovan D. Phillips. 3
   4969 Wiltshire Rd.
   North Royalton, Ohio 44133

6 483NPG
TH 2797
10/99 22527-200 NILE

DUDLEY KNOX LIBRARY
3 2768 00366734 6